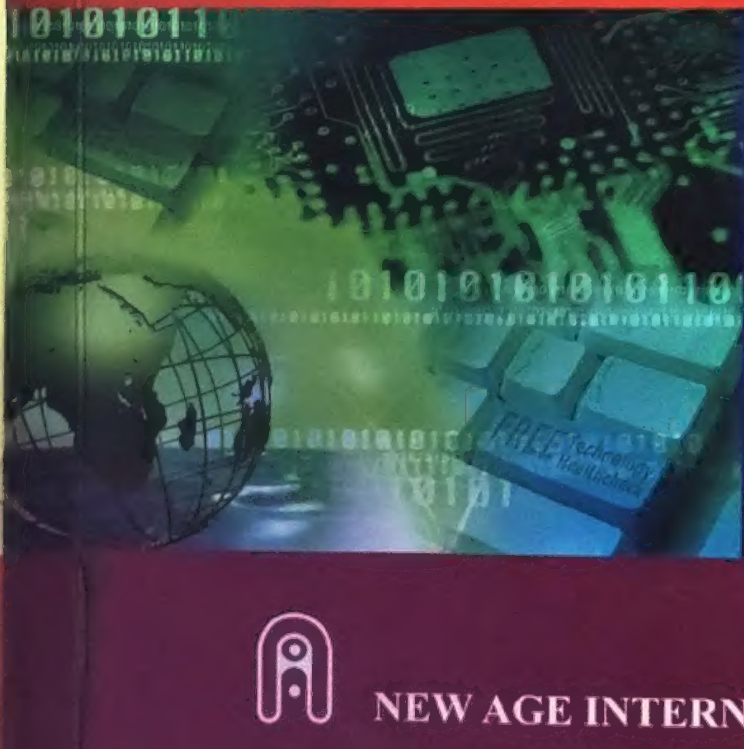
NEW AGE

SOFTWARE ENGINEERING

(THIRD EDITION)

Free CD Inside

Power Point Presentations
Given in CD Exclusively for Faculties



K.K. Aggarwal
Yogesh Singh

NEW AGE INTERNATIONAL PUBLISHERS

3rd Edition

SOFTWARE ENGINEERING

K.K. Aggarwal
Yogesh Singh

NEW AGE TECHNO PRESS

SOFTWARE ENGINEERING

Preface to the Third Edition

غامل کمپوز کنندہ

The response to the second edition of the book has been overwhelming when we look at the number of readers in every part of the country. We have received many requests to incorporate few more topics specific to the software engineering curricula of the respective universities. The feedback of readers obviously inspired us to include some of the requested topics.

Since the pedagogy of class room delivery is fast changing and power point presentations have become a way of life. To help the teachers & the students, chapter wise power point presentations have been prepared and a copy of CD is attached to this edition of the book.

Working on every new edition is like maintaining software which may include corrective, adaptive, perfective and preventive maintenance activities. As we all know, corrective maintenance refers to modifications initiated by defects in the software (mistakes in the book), adaptive maintenance includes modifying the software to match changes in the ever-changing world (modifications due to changes in curricula, changes in techniques, etc.), perfective maintenance means adding new functionality : making the product better, faster, smaller, better documented, with more functions (additions of new topics, change in style, improvement in text & figures in the book) and preventive maintenance refers to code restructuring, optimization & documentation updating (some existing topics to be rewritten, figures to be redesigned and some concepts to be redefined in the book). All four maintenance activities are in a similar manner applicable for designing a new edition of a book. As perfective maintenance effort is about 50%, in this edition too, reader's suggestions, reviews and addition of new topics consume more than 50% of the effort. We have thus tried to bring this new edition of the book in a philosophy similar to bringing out the next version of a software.

A new chapter on software certification which has become a buzzword for software community has also been added. There lies confusion about the usage and target audience for such certifications. To whom should we target: people, process, product or all the three P's. The quality of the product is dependent on the process that creates it and the people who have developed it. Therefore, quality of every segment is important for the ultimate quality of the end product.

This book is designed as a text book for the first course in software engineering for undergraduate and postgraduate students. This may also help the software professionals to implement software engineering concepts & practices. We are thankful to readers, students, teachers, researchers and practitioners whose suggestions and ideas find a place in this book. We are also thankful to Ms. Ruchika Malhotra, Research Scholar for assistance in the preparation of power point presentations.

As mentioned earlier, software maintenance is an ongoing activity for the success & stability of software. We thus simultaneously commence working for yet another edition of the book. We eagerly await suggestions, constructive criticism and new ideas for the improvement of the book from our esteemed readers.

Till then, good bye!

Prof. K.K. Aggarwal
Prof. Yogesh Singh

Preface to the Second Edition

As promised in the first edition, we have come up with the second edition of the text within four years. This edition is primarily based on suggestions from our readers and considered advice of the reviewers. The fact also remains that software engineering discipline is also maturing day by day. The increasing dependence of society on the software forces all of us to work hard to make software engineering as a stable discipline, where we can estimate development time and cost with reasonable accuracy and precision. Time is compelling us to improve software development processes in order to provide good quality maintainable software within reasonable cost and development time. As we know, quality is easy to feel but is difficult to define and impossible to measure. Hence, a quality software product means expecting a lot from the software engineering discipline, where every process is dominated by human beings.

These challenges are not only driving the industry but also making the universities to upgrade their curricula in the areas of Computer Science & Engineering, Computer Applications, Information Technology, Electronics & Communication Engineering and Electrical Engineering. The Second Edition is an attempt to bridge the gap between "What is taught in the classroom" and "What is practiced" in the industry. The concepts are discussed with the help of real life examples and numerical problems.

This book is designed as a textbook for the first course in Software Engineering for undergraduate and postgraduate students. This may also be helpful for software professionals to help them practice the software engineering concepts. We are indebted to Dr. Jitender Chhabra, Lecturer, National Institute of Technology, Kurukhshetra, Dr. Pravin Chandra, Lecturer, Guru Gobind Singh Indraprastha University, Delhi, Sh. R. K. Singh, Programme Co-ordinator, C-DAC, Noida and Ms. Arvinder Kaur, Lecturer, Guru Gobind Singh Indraprastha University for their valuable suggestions. We are extremely thankful to our students and other readers of first edition, from whom we have received more than we have given. We are also thankful to researchers and practitioners of the field whose ideas and techniques find a place in this book.

We do understand that there is nothing like a perfect product and same is true about this book. Hence we would welcome further suggestions from our readers. These suggestions shall motivate us to work on third edition of the book. Till then, good bye!

Prof. K.K. Aggarwal
Prof. Yogesh Singh

Preface to the First Edition

Developing software system is generally a quite complex and time consuming process. Moreover, the nature and complexity of software requirements have drastically changed in the last few decades and users all over the world have become much more demanding in terms of cost, schedule and quality. These three parameters, all being desirable, have an apparent contradiction at times which can only be resolved by optimum design of software using well established software engineering methodologies.

Software engineering methodologies constitute the framework that guides us in optimally developing the software systems. These frameworks define the different phases of software development, such as planning, requirements analysis, design testing and maintenance. The choice of which methodology to use in a specific development process is closely related to the size, complexity, reliability and maintainability of the software, and to the environment it is supposed to function in.

Given unlimited resources, the majority of software problems can probably be solved but the challenge confronting software developers is to produce high quality maintainable software with a finite amount of resources and to a specified schedule. This challenge forces us to adopt software engineering concepts, methodologies and practices in order to improve the software development process and thereby the product.

Keeping in view the software development potential the world over, software engineering is becoming an integral part of most of the universities' curricula in the discipline of Computer Science & Engineering, Computer Applications, Information Technology, Electronics & Communication Engineering and Electrical Engineering. By virtue of our experience with the industry, we realized that there is a wide gap between what is taught in the classroom and what is practised in the industry. In addition to theoretical concepts, sincere effort has therefore been made to bridge this gap between theory and practice in this text.

The book is intended to serve the requirements to a First Course on Software Engineering, whether at undergraduate or post-graduate level. While the students will find the text extremely useful, it will also serve the interests of the software professionals by making available to them the requisite material in one volume and help them to adapt their software development methodologies in the most suitable manner.

We are indebted to Dinesh Kumar, Pravin Chandra and Jitender Chhabra for their painstaking reading of the text. They found time in their busy schedule to not only read the text but also offer valuable suggestions for improvement. We wish to put on record the excellent services of Deepak Kumar and Sanjay Kumar for typing the draft of the manuscript. We owe a

sincere debt of gratitude to our students, from whom we have received more than we have given and to the many reasearchers and practitioners of software engineering whose ideas and techniques find a place in this book. We thank everyone at New Age International, especially Shri Anand Aswal, for their efforts in making this book a reality and that too in the most presentable form in a record time.

Being software developers, we do realise that there is nothing like a perfect product and hence we would welcome constructive comments and suggestions from our readers in order to further improve the next version (edition). Any feedback in this direction would be gratefully received and acknowledged.

Prof. K.K. Aggarwal

Prof. Yogesh Singh

Contents

5. SOFTWARE DESIGN

10. SOFTWARE CERTIFICATION 492-50

Introduction to Software Engineering 1

بطور قابل ملاحظہ

The nature and complexity of software have changed significantly in the last 30 years. In the 1970s, applications ran on a single processor, produced alphanumeric output, and received their input from a linear source. Today's applications are far more complex; typically have graphical user interface and client-server architecture. They frequently run on two or more processors, under different operating systems, and on geographically distributed machines.

Rarely, in history has a field of endeavor evolved as rapidly as software development. The struggle to stay, abreast of new technology, deal with accumulated development backlogs, and cope with people issues has become a treadmill race, as software groups work as hard as they can, just to stay in place. The initial concept of one "guru", indispensable to a project and hostage to its continued maintenance has changed. The Software Engineering Institute (SEI) and group of "gurus" advise us to improve our development process. Improvement means "ready to change". Not every member of an organization feels the need to change. It is too easy to dismiss process improvement efforts as just the latest management fad. Therein lie the seeds of conflict, as some members of a team embrace new ways of working, while others mutter "over my dead body" [WIEG94].

Therefore, there is an urgent need to adopt software engineering concepts, strategies, practices to avoid conflict, and to improve the software development process in order to deliver good quality maintainable software in time and within budget.

1.1 THE EVOLVING ROLE OF SOFTWARE

The software has seen many changes since its inception. After all, it has evolved over the period of time against all odds and adverse circumstances. Computer industry has also progressed at a break-neck speed through the computer revolution, and recently, the network revolution triggered and/or accelerated by the explosive spread of the internet and most recently the web. Computer industry has been delivering exponential improvement in price-performance, but the problems with software have not been decreasing. Software still come late, exceed budget and are full of residual faults. As per the latest IBM report, "31% of the projects get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts" [IBMG2K].

1.1.1 Some Software Failures

A major problem of software industry is its inability to develop bug free software. If software developers are asked to certify that the developed software is bug free, no software would have

ever been released. Hence, "software crisis" has become a fixture of everyday life. Many well published failures have had not only major economic impact but also become the cause of death of many human beings. Some of the failures are discussed below :

(*i*) The Y2K problem was the most crucial problem of last century. It was simply the ignorance about the adequacy or otherwise of using only last two digits of the year. The 4-digit date format, like 1964, was shortened to 2-digit format, like 64. The developers could not visualise the problem of year 2000. Millions of rupees have been spent to handle this practically non-existent problem.

(*ii*) The "star wars" program of USA produced "Patriot missile" and was used first time in Gulf war. Patriot missiles were used as a defence for Iraqi Scud missiles. The Patriot missiles failed several times to hit Scud missiles, including one that killed 28 U.S. soldiers in Dhahran, Saudi Arabia. A review team was constituted to find the reason and result was software bug. A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

(*iii*) In 1996, a US consumer group embarked on an 18-month, $1 million project to replace its customer database. The new system was delivered on time but did not work as promised, handling routine transactions smoothly but tripping over more complex ones. Within three weeks the database was shutdown, transactions were processed by hand and a new team was brought in to rebuild the system. Possible reasons for such a failure may be that the design team was over optimistic in agreeing to requirements and developers became fixated on deadlines, allowing errors to be ignored.

(*iv*) "One little bug, one big crash" of Ariane-5 space rocket, developed at a cost of $7000 M over a 10 year period. The space rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles alongwith its payload of four expensive and uninsured scientific satellites. The reason was very simple. When the guidance system's own computer tried to convert one piece of data—the sideways velocity of the rocket—from a 64-bit format to a 16-bit format; the number was too big, and an overflow error resulted after 36.7 seconds. When the guidance system shutdown, it passed control to an identical, redundant unit, which was there to provide backup in case of just such a failure. Unfortunately, the second unit had failed in the identical manner a few milliseconds before. In this case, the developers had decided that this particular velocity figure would never be large enough to cause trouble—after all, it never had been before.

(*v*) Financial software is an essential part of any company's I.T. infrastructure. However, many companies have experienced failures in their accounting system due to faults in the software itself. The failures ranged from producing the wrong information to the complete system crashing. There is a widespread dissatisfaction over the quality of the financial software. Even if a system only gives incorrect information, this may have an adverse impact on confidence.

(*vi*) Charles C. Mann shared his views about windows XP through his article in technology review [MANN02] as :

> "Microsoft released windows XP on october 25, 2001 and on the same day, which may be a record, the company posted 18 megabytes of patches on its website for bug fixes, compatibility updates, and enhancements. Two patches were intended to fix important

security holes. (One of them did ; the other patch did not work). Microsoft advised (still advises) users to back up critical files before installing patches".

This situation is quite embarrasing and clearly explains the sad state of affairs of present software companies. The developers were either too rushed or too careless to fix obvious defects.

We may keep on discussing the history of software failures which have played with human safety and caused the projects failures in the past.

1.1.2 No Silver Bullet

We have discussed some of the software failures. Is there any light ? Or same scenario would continue for many years. When automobile engineers discuss the cars in the market, they do not say that cars today are no better than they were ten or fifteen years ago. The same is true for aeronautical engineers ; no body says that Boeing or Airbus does not make better quality planes as compared to their previous decade planes. Civil engineers also do not show their anxieties over the quality of today's structures over the structures of last decade. Every-one feels that things are improving day by day. But software, also, seems different. Many software engineers believe that software quality is NOT improving. If anything they say, "it is getting worse".

As we all know, the hardware cost continues to decline drastically. However, there are desperate cries for a silver bullet-something to make software costs drop as rapidly as computer hardware costs do. But as we look to the horizon of a decade, we see no silver bullet. There is no single development, either in technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.

Inventions in electronic design through transistors and large scale integration has significantly affected the cost, performance and reliability of the computer hardware. No other technology, since civilization began, has seen six orders of magnitude in performance-price gain in 30 years. The progress in software technology is not that rosy due to certain difficulties with this technology. Some of the difficulties are complexity, changeability and invisibility.

The hard part of building software is the specification, design and testing of this conceptual construct, not the labour of representing it and testing the correctness of representation. We still make syntax errors, to be sure, but they are trivial as compared to the conceptual errors (logic errors) in most systems. That is why, building software is always hard and there is inherently no silver bullet.

Many people (especially CASE tool vendors) believe that CASE (Computer Aided Software Engineering) tools represent the so-called silver bullet that would rescue the software industry from the software crisis. Many companies have used these tools and spent large sums of money, but results were highly unsatisfactory, we learnt the hard way that there is no such thing as a silver bullet [BROO87].

The software is evolving day by day and its impact is also increasing on every facet of human life. We cannot imagine a day without using cell phones, logging on to the internet, sending e-mails, watching television and so on. All these activities are dependent on software and software bugs exist nearly everywhere. The blame for these bugs goes to software companies that rush products to market without adequately testing them. It belongs to software

developers who could not understand the importance of detecting and removing faults before the customer experiences them as failures. It belongs to the legal system that has given a free pass to software developers on bug related damages. It also belongs to universities and other higher educational institutions that stress on programming over software engineering principles and practices.

1.2 WHAT IS SOFTWARE ENGINEERING?

Software has become critical to advancement in almost all areas of human endeavour. The art of programming only is no longer sufficient to construct large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products.

Software engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the product.

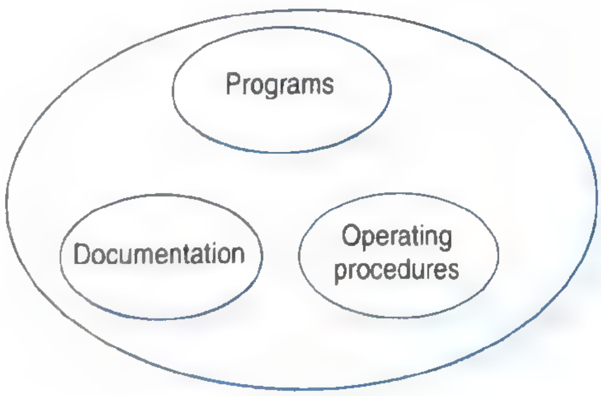
1.2.1 Definition

At the first conference on software engineering in 1968, Fritz Bauer [FRIT68] defined software engineering as *"The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines"*. Stephen Schach [SCHA90] defined the same as *"A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements"*.

Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

1.2.2 Program Versus Software

Software is more than programs. It consists of programs, documentation of any facet of the program and the procedures used to setup and operate the software system. The components of the software systems are shown in Fig. 1.1.



Software = Program + Documentation + Operating Procedures

Fig. 1.1: Components of software

Any program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared. Program is a combination of source code and object code. Documentation consists of different types of manuals as shown in Fig. 1.2.

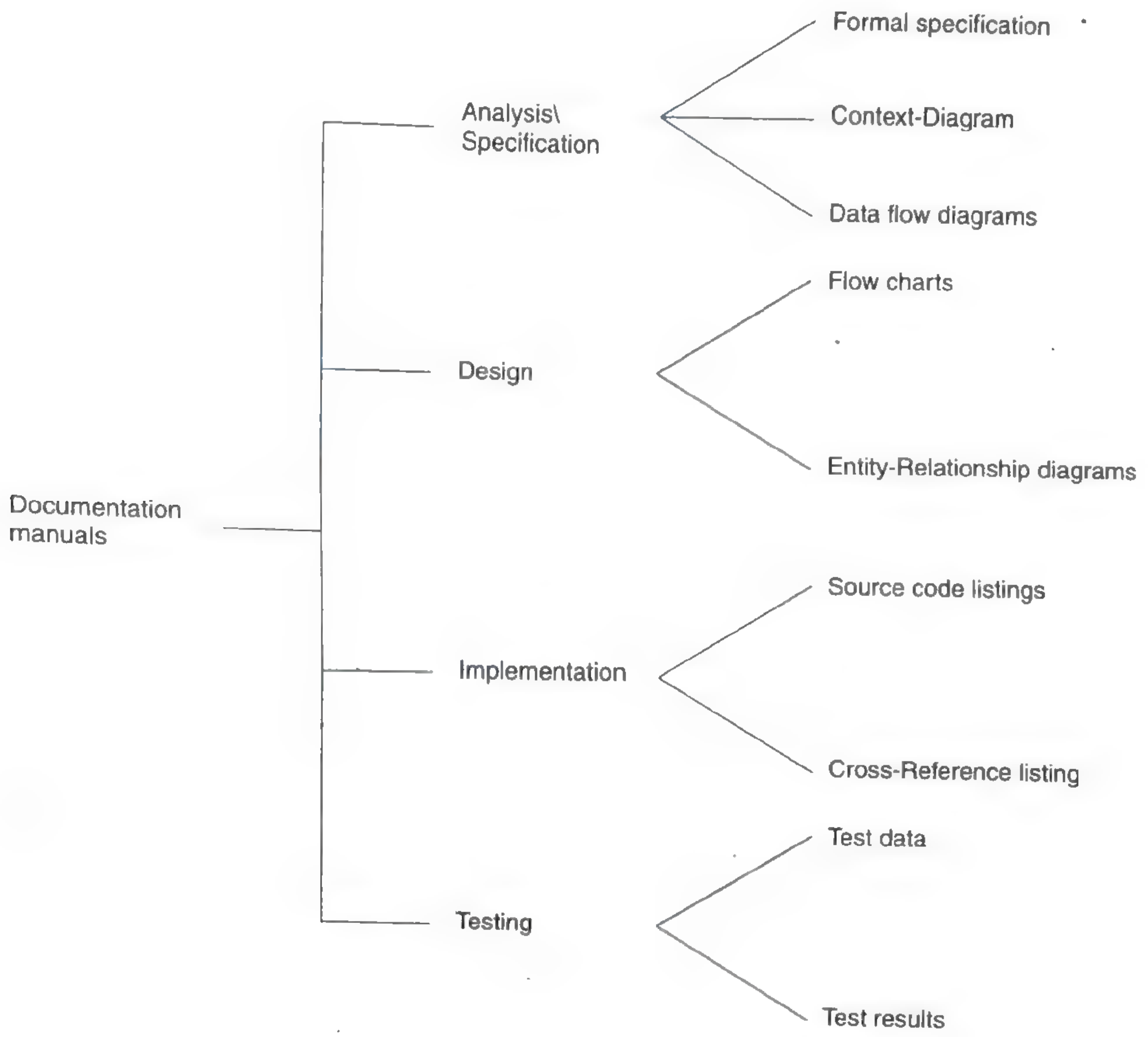


Fig. 1.2: List of documentation manuals

Operating procedures consist of instructions to setup and use the software system and instructions on how to react to system failure. List of operating procedure manuals/documents is given in Fig. 1.3.

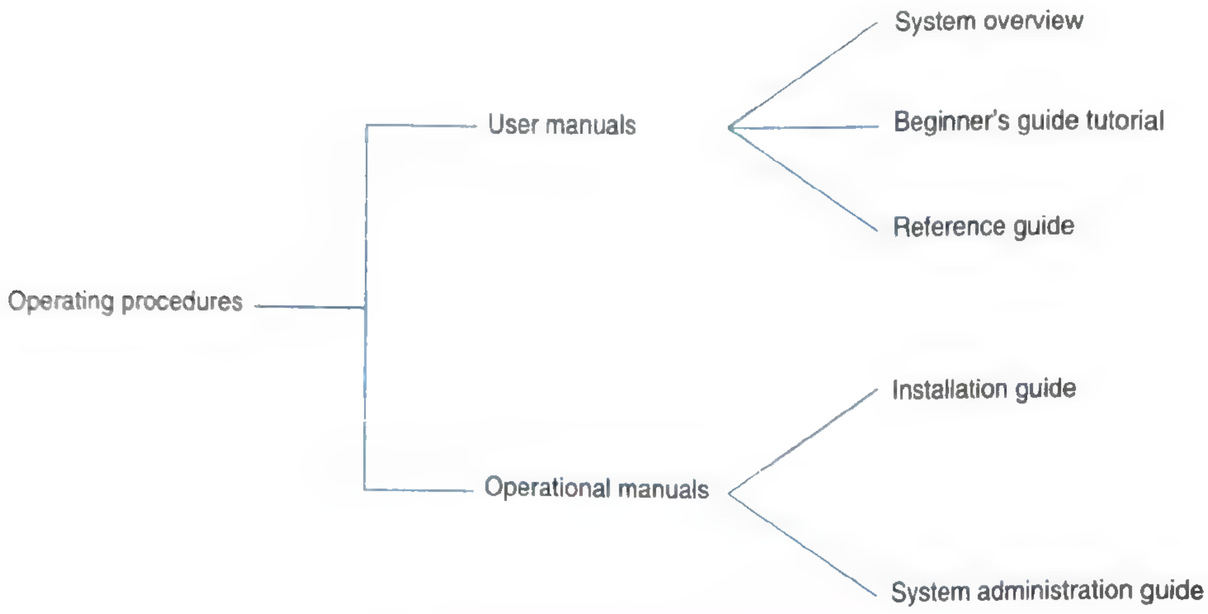


Fig. 1.3: List of operating procedure manuals

1.2.3 Software Process

The software process is the way in which we produce software. This differs from organization to organization. Surviving in the increasingly competitive software business requires more than hiring smart, knowledgeable developers and buying the latest development tools. We also need to use effective software development processes, so that developers can systematically use the best technical and managerial practices to successfully complete their projects. Many software organizations are looking at software process improvement as a way to improve the quality, productivity, predictability of their software development, and maintenance efforts [WIEG96].

It seems straight forward, and the literature has a number of success stories of companies that substantially improved their software development and project management capabilities. However, many other organizations do not manage to achieve significant and lasting improvements in the way they conduct their projects. Here we discuss few reasons why is it difficult to improve software process [HUMP89, WIEG99] ?

1. **Not enough time:** Unrealistic schedules leave insufficient time to do the essential project work. No software groups are sitting around with plenty of spare time to devote to exploring what is wrong with their current development processes and what they should be doing differently. Customers and senior managers are demanding more software, of higher quality in minimum possible time. Therefore, there is always a shortage of time. One consequence is that software organizations may deliver release 1.0 on time, but then they have to ship release 1.01 almost immediately thereafter to fix the recently discovered bugs.

2. **Lack of knowledge:** A second obstacle to widespread process improvement is that many software developers do not seem to be familiar with industry best practices. Normally, software developers do not spend much time reading the literature to find out about the best-known ways of software development. Developers may buy books on Java, Visual Basic or ORACLE, but do not look for anything about process, testing or quality on their bookshelves.

The industry awareness of process improvement frameworks such as the capability maturity model and ISO 9001 for software (discussed in Chapter 7) have grown in recent years, but effective and sensible application still is not that common. Many recognized best practices available in literature simply are not in widespread use in the software development world.

3. **Wrong motivations:** Some organizations launch process improvement initiatives for the wrong reasons. May be an external entity, such as a contractor, demanded that the development organization should achieve CMM level X by date Y. Or perhaps a senior manager learned just enough about the CMM and directed his organization to climb on the CMM bandwagon.

The basic motivation for software process improvement should be to make some of the current difficulties we experience on our projects to go away. Developers are rarely motivated by seemingly arbitrary goals of achieving a higher maturity level or an external certification (ISO 9000) just because someone has decreed it. However, most people should be motivated by the prospect of meeting their commitments, improving customer satisfaction, and delivering excellent products that meet customer expectations. The developers have resisted many process improvement initiatives when they were directed to do "the CMM thing", without a clear explanation of the reasons why improvement was needed and the benefits the team expected to achieve.

4. **Insufficient commitment:** Many times, the software process improvement fails, despite best of intentions, due to lack of true commitment. It starts with a process assessment but fails to follow through with actual changes. Management sets no expectations from the development community around process improvement; they devote insufficient resources, write no improvement plan, develop no roadmap, and pilot no new processes.

The investment we make in process improvement will not have an impact on current productivity; because the time we spend developing better ways to work tomorrow is not available for today's assignment. It can be tempting to abandon the effort when skeptics see the energy they want to be devoted to immediate demands being siphoned off in the hope of a better future (Fig. 1.4). Software organizations should not give up, but should take motivation from the very real, long-term benefits that many companies (including Motorola, Hewlett-Packard, Boeing, Microsoft etc.) have enjoyed from sustained software process improvement initiatives. Improvements will take place over time and organizations should not expect and promise miracles [WIEG2K] and should always remember the learning curve.

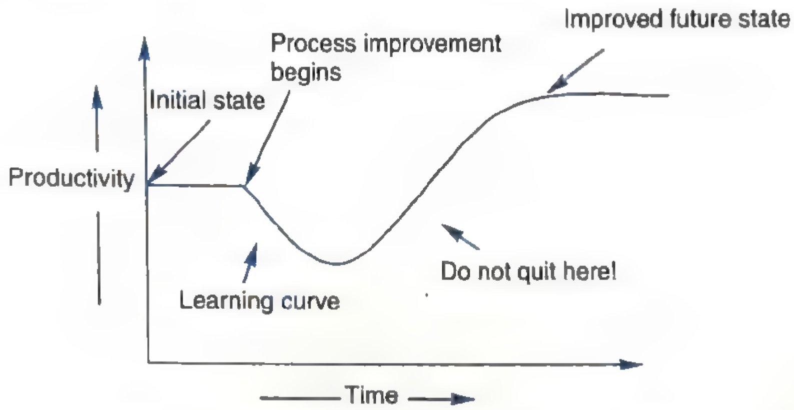


Fig. 1.4: The process improvement learning curve

1.2.4 Software Characteristics

The software has a very special characteristic *e.g.*, "it does not wear out". Its behaviour and nature is quite different than other products of human life. A comparison with one such case, *i.e.*, constructing a bridge vis-a-vis writing a program is given in Table 1.1. Both activities require different processes and have different characteristics.

Table 1.1: A comparison of constructing a bridge and writing a program

Sr. No.	*Constructing a bridge*	*Writing a program*
1.	The problem is well understood.	Only some parts of the problem are understood, others are not.
2.	There are many existing bridges.	Every program is different and designed for special applications.
3.	The requirements for a bridge typically do not change much during construction.	Requirements typically change during all phases of development.
4.	The strength and stability of a bridge can be calculated with reasonable precision.	Not possible to calculate correctness of a program with existing methods.
5.	When a bridge collapses, there is a detailed investigation and report.	When a program fails, the reasons are often unavailable or even deliberately concealed.
6.	Engineers have been constructing bridges for thousands of years.	Developers have been writing programs for 50 years or so.
7.	Materials (wood, stone, iron, steel) and techniques (making joints in wood, carving stone, casting iron) change slowly.	Hardware and software changes rapidly.

Some of the important characteristics are discussed below:

(i) **Software does not wear out:** There is a well-known "bath tub curve" in reliability studies for hardware products. The curve is given in Fig. 1.5. The shape of the curve is like "bath tub"; and is known as bath tub curve.

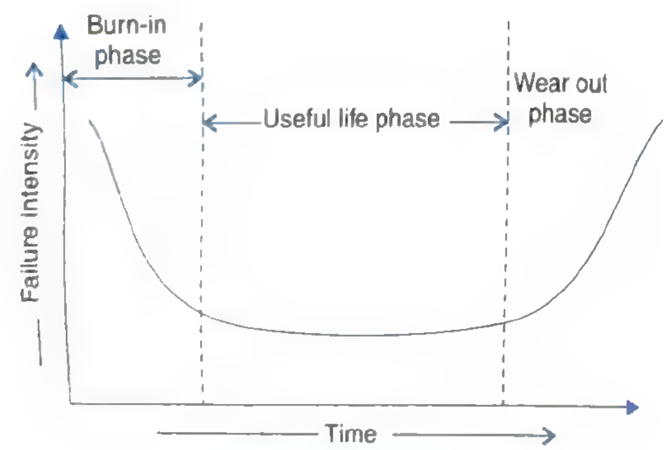


Fig. 1.5: Bath tub curve

There are three phases for the life of a hardware product. Initial phase is burn-in phase, where failure intensity is high. It is expected to test the product in the industry before delivery. Due to testing and fixing faults, failure intensity will come down initially and may stabilise after certain time. The second phase is the useful life phase where failure intensity is approximately constant and is called useful life of a product. After few years, again failure intensity will increase due to wearing out of components. This phase is called wear out phase. We do not have this phase for the software as it does not wear out. The curve for software is given in Fig. 1.6.

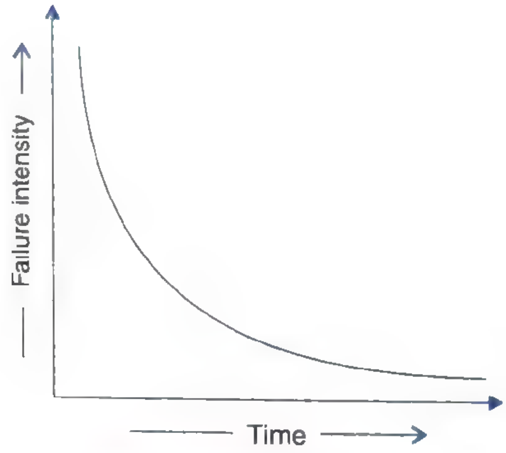


Fig. 1.6: Software curve

Important point is software becomes reliable overtime instead of wearing out. It becomes obsolete, if the environment for which it was developed, changes. Hence software may be retired due to environmental changes, new requirements, new expectations, etc.

(*ii*) **Software is not manufactured:** The life of a software is from concept exploration to the retirement of the software product. It is one time development effort and continuous maintenance effort in order to keep it operational. However, making 1000 copies is not an issue and it does not involve any cost. In case of hardware product, every product costs us due to raw material and other processing expenses. We do not have assembly line in software development. Hence it is not manufactured in the classical sense.

(*iii*) **Reusability of components:** If we have to manufacture a TV, we may purchase picture tube from one vendor, cabinet from another, design card from third and other electronic components from fourth vendor. We will assemble every part and test the product thoroughly to produce a good quality TV. We may be required to manufacture only a few components or no component at all. We purchase every unit and component from the market and produce the finished product. We may have standard quality guidelines and effective processes to produce a good quality product.

In software, every project is a new project. We start from the scratch and design every unit of the software product. Huge effort is required to develop a software which further increases the cost of the software product. However, effort has been made to design standard components that may be used in new projects. Software reusability has introduced another area and is known as component based software engineering.

Hence developers can concentrate on truly innovative elements of design, that is, the parts of the design that represent something new. As explained earlier, in the hardware world, component reuse is a natural part of the engineering process. In software, there is only a humble beginning like graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.

(iv) **Software is flexible:** We all feel that software is flexible. A program can be developed to do almost anything. Sometimes, this characteristic may be the best and may help us to accomodate any kind of change. However, most of the times, this "almost anything" characteristic has made software development difficult to plan, monitor and control. This unpredictability is the basis of what has been referred to for the past 30 years as the "Software Crisis".

1.3 THE CHANGING NATURE OF SOFTWARE

Software has become integral part of most of the fields of human life. We name a field and we find the usage of software in that field. Software applications are grouped in to eight areas for convenience as shown in Fig. 1.7.

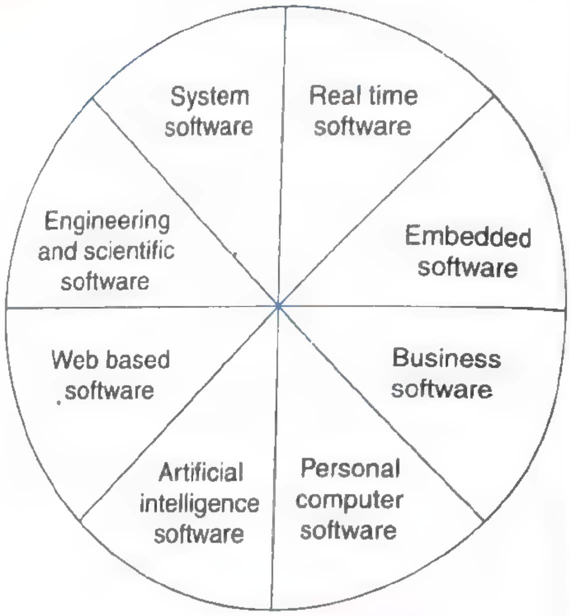


Fig. 1.7: Software applications

(i) **System software:** Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a collection of programs to provide service to other programs.

(ii) **Real time software:** These software are used to monitor, control and analyze real world events as they occur. An example may be software required for weather forcasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forcast the weather.

(iii) **Embedded software:** This type of software is placed in "Read-Only-Memory (ROM)" of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software.

(iv) **Business software:** This is the largest application area. The software designed to process business applications is called business software. Business software could be payroll, file monitoring system, employee management, account management. It may also be a data warehousing tool which helps us to take decisions based on available data. Management information system, enterprise resource planning (ERP) and such other software are popular examples of business software.

(v) **Personal computer software:** The software used in personal computers are covered in this category. Examples are word processors, computer graphics, multimedia and animating

tools, database management, computer games etc. This is a very upcoming area and many big organisations are concentrating their effort here due to large customer base.

(*vi*) **Artificial intelligence software:** Artificial Intelligence software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis [PRESOI]. Examples are expert systems, artificial neural network, signal processing software etc.

(*vii*) **Web based software:** The software related to web applications come under this category. Examples are CGI, HTML, Java, Perl, DHTML etc.

(*viii*) **Engineering and scientific software:** Scientific and engineering application software are grouped in this category. Huge computing is normally required to process data. Examples are CAD/CAM package, SPSS, MATLAB, Engineering Pro, Circuit analyzers etc.

The expectations from software are increasing in modern civilisation. Software of any of the above groups, has a specialised role to play. Customers and development organisations are desiring more features which may not be always possible to provide. Another trend has emerged to provide source code to the customers and organisations so that they can make modifications for their needs. This trend is particularly visible in infrastructure software like data bases, operating systems, compilers etc. Software where source codes are available, are known as open source. Organisations can develop software applications around such source codes. Some of the examples of "open source software" are LINUX, MySQL, PHP, open office, Apache web server etc. Open source software has risen to great prominence. We may say that these are the programs whose licenses give users the freedom to run the program for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program without paying royalties to original developers. Whether open source software are better than proprietary software ? Answer is not easy. Both schools of thought are in the market. However, popularity of many open source software give confidence to every user. They may also help us to develop small business applications at low cost.

1.4 SOFTWARE MYTHS

There are number of myths associated with software development community. Some of them really affect the way, in which software development should take place. In this section, we list few myths, and discuss their applicability to standard software development [PIER99, LEVE95].

1. **Software is easy to change:** It is true that source code files are easy to edit, but that is quite different than saying that software is easy to change. This is deceptive precisely because source code is so easy to alter. But making changes without introducing errors is extremely difficult, particularly in organizations with poor process maturity. Every change requires that the complete system be re-verified. If we do not take proper care, this will be an extremely tedious and expensive process.

2. **Computers provide greater reliability than the devices they replace:** It is true that software does not fail in the traditional sense. There are no limits to how many times a given piece of code can be executed before it "wears out". In any event, the simple expression of this myth is that our general ledgers are still not perfectly accurate, even though they have

been computerized. Back in the days of manual accounting systems, human error was a fact of life. Now, we have software error as well.

3. **Testing software or 'proving' software correct can remove all the errors:** Testing can only show the presence of errors. It cannot show the absence of errors. Our aim is to design effective test cases in order to find maximum possible errors. The more we test, the more we are confident about our design.

4. **Reusing software increases safety:** This myth is particularly troubling because of the false sense of security that code re-use can create. Code re-use is a very powerful tool that can yield dramatic improvement in development efficiency, but it still requires analysis to determine its suitability and testing to determine if it works.

5. **Software can work right the first time:** If we go to an aeronautical engineer, and ask him to build a jet fighter craft, he will quote us a price. If we demand that it is to be put in production without building a prototype, he will laugh and may refuse the job. Yet, software engineers are often asked to do precisely this sort of work, and they often accept the job.

6. **Software can be designed thoroughly enough to avoid most integration problems:** There is an old saying among software designers: "Too bad, there is no complier for specifications": This points out the fundamental difficulty with detailed specifications. They always have inconsistencies, and there is no computer tool to perform consistency checks on these. Therefore, special care is required to understand the specifications, and if there is an ambiguity, that should be resolved before proceeding for design.

7. **Software with more features is better software:** This is, of course, almost the opposite of the truth. The best, most enduring programs are those which do one thing well.

8. **Addition of more software engineers will make up the delay:** This is not true in most of the cases. By the process of adding more software engineers during the project, we may further delay the project. This does not serve any purpose here, although this may be true for any civil engineering work.

9. **Aim is to develop working programs:** The aim has been shifted from developing working programs to good quality, maintainable programs. Maintaining software has become a very critical and crucial area for software engineering community.

This list is endless. These myths, poor quality of software, increasing cost and delay in the delivery of the software have been the driving forces behind the emergence of software engineering as a discipline. In addition, following are the contributing factors:

- Change in ratio of hardware to software costs
- Increasing importance of maintenance
- Advances in software techniques
- Increased demand for software
- Demand for larger and more complex software systems.

1.5 SOME TERMINOLOGIES

Some terminologies are discussed in this section which are frequently used in the field of Software Engineering.

1.5.1 Deliverables and Milestones

Different deliverables are generated during software development. The examples are source code, user manuals, operating procedure manuals etc.

The milestones are the events that are used to ascertain the status of the project. Finalisation of specification is a milestone. Completion of design documentation is another milestone. The milestones are essential for project planning and management.

1.5.2 Product and Process

Product: What is delivered to the customer, is called a product. It may include source code, specification document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.

Process: Process is the way in which we produce software. It is the collection of activities that leads to (a part of) a product. An efficient process is required to produce good quality products.

If the process is weak, the end product will undoubtedly suffer, but an obsessive over-reliance on process is also dangerous.

1.5.3 Measures, Metrics and Measurement

The terms measures, metrics and measurement are often used interchangeably. It is interesting to understand the difference amongst these. A measure provides a quantitative indication of the extent, dimension, size, capacity, efficiency, productivity or reliability of some attributes of a product or process.

Measurement is the act of evaluating a measure. A metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute. Pressman [PRESO2] explained this very effectively with an example as given below:

"When a single data point has been collected (*e.g.*, the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (*e.g.*, a number of module reviews are investigated to collect measures of the number of errors in each module). A software metric relates the individual measures in some way (*e.g.*, the average number of errors found per review)."

Hence we collect measures and develop metrics to improve the software engineering practices.

1.5.4 Software Process and Product Metrics

Software metrics are used to quantitatively characterise different aspects of software process or software products. Process metrics quantify the attributes of software development process and environment; whereas product metrics are measures for the software product. Examples of process metrics include productivity, quality, failure rate, efficiency etc. Examples of product metrics are size, reliability, complexity, functionality etc.

1.5.5 Productivity and Effort

Productivity is defined as the rate of output, or production per unit of effort, *i.e.*, the output achieved with regard to the time taken but irrespective of the cost incurred. Hence, there are two issues for deciding the unit of measure

(*i*) quantity of output

(*ii*) period of time.

In software, one of the measure for quantity of output is lines of code (LOC) produced. Time is measured in days or months.

Hence most appropriate unit of effort is Person Months (PMs), meaning thereby number of persons involved for specified months. So, productivity may be measured as LOC/PM (lines of code produced/person month).

1.5.6 Module and Software Components

There are many definitions of the term module. They range from "a module is a FORTRAN subroutine" to "a module is an Ada Package", to "Procedures and functions of PASCAL and C", to "C++ Java classes" to "Java packages" to "a module is a work assignment for an individual developer". All these definitions are correct. The term subprogram is also used sometimes in place of module.

There are many definitions of software components. A general definition given by Alan W. Brown [BROW2K] is:

"An independently deliverable piece of functionality providing access to its services through interfaces".

Another definition from unified modeling language (UML) [OMG2K] is:

"A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces".

Hence, a reusable module is an independent and deliverable software part that encapsulates a functional specification and implementation for reuse by a third party.

However, a reusable component is an independent, deployable, and replaceable software unit that is reusable by a third party based on unit's specification, implementation, and well defined contracted interfaces.

1.5.7 Generic and Customised Software Products

The software products are divided in two categories:

(*i*) Generic products

(*ii*) Customised products.

Generic products are developed for anonymous customers. The target is generally the entire world and many copies are expected to be sold. Infrastructure software like operating systems, compilers, analysers, word processors, CASE tools etc. are covered in this category.

The customised products are developed for particular customers. The specific product is designed and developed as per customer requirements. Most of the development projects (say about 80%) come under this category.

1.6 ROLE OF MANAGEMENT IN SOFTWARE DEVELOPMENT

The management of software development is heavily dependent on four factors: People, Product, Process, and Project. Order of dependency is as shown in Fig. 1.8.

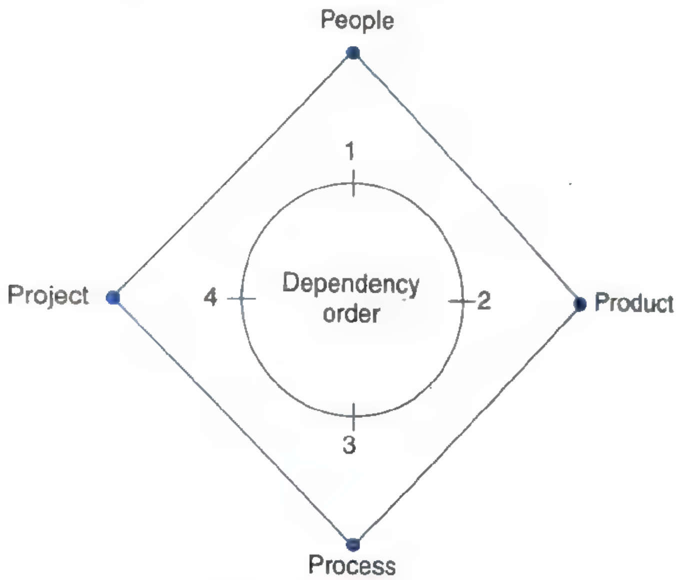


Fig. 1.8: Factors of management dependency (from People to Project)

Software development is a people centric activity. Hence, success of the project is on the shoulders of the people who are involved in the development.

1.6.1 The People

Software development requires good managers. The managers, who can understand the psychology of people and provide good leadership. A good manager cannot ensure the success of the project, but can increase the probability of success. The areas to be given priority are: proper selection, training, compensation, career development, work culture etc.

Managers face challenges. It requires mental toughness to endure inner pain. We need to plan for the best, be prepared for the worst, expect surprises, but continue to move forward anyway. Charles Maurice once rightly said "I am more afraid of an army of one hundred sheep led by a lion than an army of one hundred lions led by a sheep".

Hence, manager selection is most crucial and critical. After having a good manager, project is in safe hands. It is the responsibility of a manager to manage, motivate, encourage, guide and control the people of his/her team.

1.6.2 The Product

What do we want to deliver to the customer? Obviously, a product; a solution to his/her problems.

Hence, objectives and scope of work should be defined clearly to understand the requirements. Alternate solutions should be discussed. It may help the managers to select a "best" approach within constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces etc. Without well defined requirements, it may be

impossible to define reasonable estimates of the cost, development time and schedule for the project.

1.6.3 The Process

The process is the way in which we produce software. It provides the framework from which a comprehensive plan for software development can be established. If the process is weak, the end product will undoubtedly suffer. There are many life cycle models and process improvements models. Depending on the type of project, a suitable model is to be selected. Now-a-days CMM (Capability Maturity Model) has become almost a standard for process framework. The process priority is after people and product, however, it plays very critical role for the success of the project. A small number of framework activities are applicable to all software projects, regardless of their size and complexity. A number of different task sets, tasks, milestones, work products, and quality assurance points, enable the framework activities to be adopted to the characteristics of the project and the requirements of the project team.

1.6.4 The Project

A proper planning is required to monitor the status of development and to control the complexity. Most of the projects are coming late with cost overruns of more than 100%. In order to manage a successful project, we must understand what can go wrong and how to do it right. We should define concrete requirements (although very difficult) and freeze these requirements. Changes should not be incorporated to avoid software surprises. Software surprises are always risky and we should minimise them. We should have a planning mechanism to give warning before the occurrence of any surprise.

All four factors (People, Product, Process and Project) are important for the success of the project. Their relative importance helps us to organise development activities in more scientific and professional way.

Software Engineering has become very important discipline of study, practice and research. All are working hard to minimise the problems and to meet the objective of developing good quality maintainable software that is delivered on time, within budget, and also satisfies the requirements. With all cries and dissatisfaction, discipline is improving and maturing day by day. New solutions are being provided in the niche areas and encouraging results are being observed. We do feel that with in couple of years, situation is bound to improve and software engineering shall be a stable and mature discipline.

REFERENCES

[BOEH89] Boehm B., "*Risk Management*", IEEE Computer Society Press, 1989.

[BROO87] Brooks F.P., "*No Silver Bullet : Essence and Accidents of Software Engineering*", IEEE Computer, 10—19, April, 1987.

[BROW2K] Brown A.W., "*Large Scale, Component based Development*", Englewood cliffs, NJ, PH, 2000.

[FRIT68] Bauer, Fritz et al., "*Software Engineering: A Report on a Conference Sponsored by NATO Science Committee*", NATO, 1968.

[HUMP89] Humphrey W.S., "*Managing the Software Process*", Addison-Wesley Pub. Co., Reading, Massachusetts, USA, 1989.

[IBMG2K] IBM Global Services India Pvt. Ltd., Golden Tower, Airport Road, Bangalore, Letter of Bindu Subramani, Segment Manager—Corporate Training, March 9, 2000.

[LEVE95] Leveson N.G., "*Software, System Safety and Computers*", Addison Wesley, 1995.

[MANN02] Charles C.Mann, "*Why Software is so Bad*", Technology Review, www. technologyreview.com. 2002.

[OMG2K] OMG Unified Modelling Language Specification, Version 1.4, Object Mangement Group, 2000.

[PRES02] Pressman R., "*Software Engineering*", McGraw Hill, 2002.

[PIER99] Piersal K., "*Amusing Software Myths*", www. bejeeber.org/Software-myths.html, 1999.

[SCHA90] Schach, Stephen, "*Software Engineering*", Vanderbilt University, Aksen Association, 1990.

[WIEG2K] Wiegers K.E., "*Stop Promising Miracles*" Software Development Magazine, February, 2000.

[WIEG94] Wiegers K.E., "*Creating a Software Engineering Culture*", Software Development Magazine, July, 1994.

[WIEG96] Wiegers K.E., "*Software Process Improvement: Ten Traps to Avoid*", Software Development Magazine, May, 1996.

[WIEG99] Wiegers K.E., "*Why is Process Improvement So Hard*", Software Development Magazine, February, 1999.

MULTIPLE CHOICE QUESTIONS

Note : Select most appropriate answer of the following questions.

1.1. Software is

(*a*) superset of programs (*b*) subset of programs

(*c*) set of programs (*d*) none of the above.

1.2. Which is NOT the part of operating procedure manuals?

(*a*) user manuals (*b*) operational manuals

(*c*) documentation manuals (*d*) installation manuals.

1.3. Which is NOT a software characteristic?

(*a*) software does not wear out (*b*) software is flexible

(*c*) software is not manufactured (*d*) software is always correct.

1.4. Product is

(*a*) deliverables (*b*) user expectations

(*c*) organisation's effort in development (*d*) none of the above.

1.5. To produce a good quality product, process should be

(*a*) complex (*b*) efficient

(*c*) rigorous (*d*) none of the above.

1.6. Which is not a product metric?

(*a*) size (*b*) reliability

(*c*) productivity (*d*) functionality.

1.7. Which is not a process metric?
(a) productivity (b) functionality
(c) quality (d) efficiency.

1.8. Effort is measured in terms of:
(a) person-months (b) rupees
(c) persons (d) months.

1.9. UML stands for
(a) uniform modeling language (b) unified modeling language
(c) unit modeling language (d) universal modeling language.

1.10. An independently deliverable piece of functionality providing access to its services throug interfaces is called
(a) software measurement (b) software composition
(c) software measure (d) software component.

1.11. Infrastructure software are covered under
(a) generic products (b) customised products
(c) generic and customised products (d) none of the above.

1.12. Management of software development is dependent on
(a) people (b) product
(c) process (d) all of the above.

1.13. During software development, which factor is most crucial?
(a) people (b) product
(c) process (d) project.

1.14. Program is
(a) subset of software (b) super set of software
(c) software (d) none of the above.

1.15. Milestones are used to
(a) know the cost of the project (b) know the status of the project
(c) know user expectations (d) none of the above.

1.16. The term module used during design phase refers to
(a) function (b) procedure
(c) sub program (d) all of the above.

1.17. Software consists of
(a) set of instructions + operating system
(b) programs + documentation + operating procedures
(c) programs + hardware manuals (d) set of programs.

1.18. Software engineering approach is used to achieve:
(a) better performance of hardware (b) error free software
(c) reusable software (d) quality software product.

1.19. Concepts of software engineering are applicable to
(a) fortran language only (b) pascal language only
(c) 'C' language only (d) all of the above.

1.20. CASE Tool is

(*a*) computer Aided Software Engineering (*b*) component Aided Software Engineering

(*c*) constructive Aided Software Engineering (*d*) computer Analysis Software Engineering.

EXERCISE

1.1. Why is the primary goal of software development now shifting from producing good quality software to good quality maintainable software?

1.2. List the reasons for the "software crisis"? Why are CASE tools not normally able to control it?

1.3. "The software crisis is aggravated by the progress in hardware technology?" Explain with examples.

1.4. What is software crisis? Was Y2K a software crisis?

1.5. What is the significance of software crisis in reference to software engineering discipline.

1.6. How are software myths affecting software process? Explain with the help of examples.

1.7. State the difference between program and software. Why have documents and documentation become very important?

1.8. What is software engineering? Is it an art, craft or a science? Discuss.

1.9. What is the aim of software engineering? What does the discipline of software engineering discuss?

1.10. Define the term "Software Engineering". Explain the major differences between software engineering and other traditional engineering disciplines.

1.11. What is software process? Why is it difficult to improve it?

1.12. Describe the characteristics of software contrasting it with the characteristics of hardware.

1.13. Write down the major characteristics of a software. Illustrate with a diagram that the software does not wear out.

1.14. What are the components of a software? Discuss how a software differs from a program.

1.15. Discuss major areas of the applications of the software.

1.16. Is software a product or process? Justify your answer with examples.

1.17. Differentiate between the followings

(*i*) Deliverables and milestones (*ii*) Product and process

(*iii*) Measures, metrics and measurement

1.18. What is software metric? How is it different from software measurement?

1.19. Discuss software process and product metrics with the help of examples.

1.20. What is productivity? How is it related to effort? What is the unit of effort?

1.21. Differentiate between module and software component.

1.22. Distinguish between generic and customised software products. Which one has larger share of market and why?

1.23. Describe the role of management in software development with the help of examples.

1.24. What are various factors of management dependency in software development? Discuss each factor in detail.

1.25. What is more important: Product or process? Justify your answer.

2 Software Life Cycle Models

The ultimate objective of software engineering is to produce good quality maintainable software within reasonable time frame and at affordable cost. This is achievable only if we have matured processes to produce it. For a mature process, it should be possible to determine in advance how much time and effort will be required to produce the final product. This can only be done using data from past experience, which requires that we must measure the software process.

Software development organizations follow some process when developing a software product. In immature organizations, the process is usually not written down. In mature organizations, the process is in writing and is actively managed. A key component of any software development process is the life cycle model on which the process is based. The particular life cycle model can significantly affect overall life cycle costs associated with a software product [RAKI97]. Life cycle of the software starts from concept exploration and ends at the retirement of the software.

In the IEEE standard Glossary of software Engineering Terminology, the software life cycle is:

"The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirement phase, design phase, implementation phase, test phase, installation and check out phase, operation and maintenance phase, and sometimes retirement phase".

A software life cycle model is a particular abstraction that represents a software life cycle. A software life cycle model is often called a software development life cycle (SDLC). A variety of life cycle models have been proposed and are based on tasks involved in developing and maintaining software. Few well known life cycles models are discussed in this chapter.

2.1 BUILD AND FIX MODEL

Sometimes a product is constructed without specifications or any attempt at design. Instead, the developer simply builds a product that is reworked as many times as necessary to satisfy the client [SCHA96].

This is an adhoc approach and not well defined. Basically, it is a simple two-phase model. The first phase is to write code and the next phase is to fix it as shown in Fig. 2.1. Fixing in this context may be error correction or addition of further functionality [TAKA96].

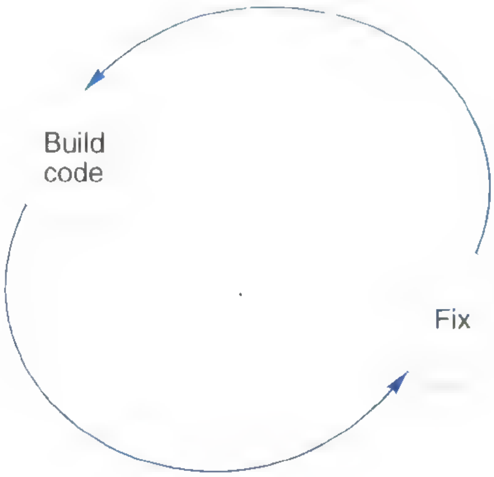


Fig. 2.1: Build and fix model

Although this approach may work well on small programming exercises 100 or 200 lines long, this model is totally unsatisfactory for software of any reasonable size. Code soon becomes unfixable and unenhanceable. There is no room for design or any aspect of development process to be carried out in a structured or detailed way. The cost of the development using this approach is actually very high as compared to the cost of a properly specified and carefully designed product. In addition, maintenance of the product can be extremely difficult without specification or design documents.

2.2 THE WATERFALL MODEL

The most familiar model is the waterfall model, which is given in Fig. 2.2. This model has five phases: Requirements analysis and specification, design, implementation and unit testing, integration and system testing and operation and maintenance. The phases always occur in this order and do not overlap. The developer must complete each phase before the next phase begins. This model is named "Waterfall Model", because its diagrammatic representation resembles a cascade of waterfalls.

1. **Requirement analysis and specification phase:** The goal of this phase is to understand the exact requirements of the customer and to document them properly. This activity is usually executed together with the customer, as the goal is to document all functions, performance and interfacing requirements for the software. The requirements describe the "what" of a system, not the "how". This phase produces a large document, written in a natural language, contains a description of what the system will do without describing how it will be done. The resultant document is known as software requirement specification (SRS) document.

The SRS document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure to implement the contracted system.

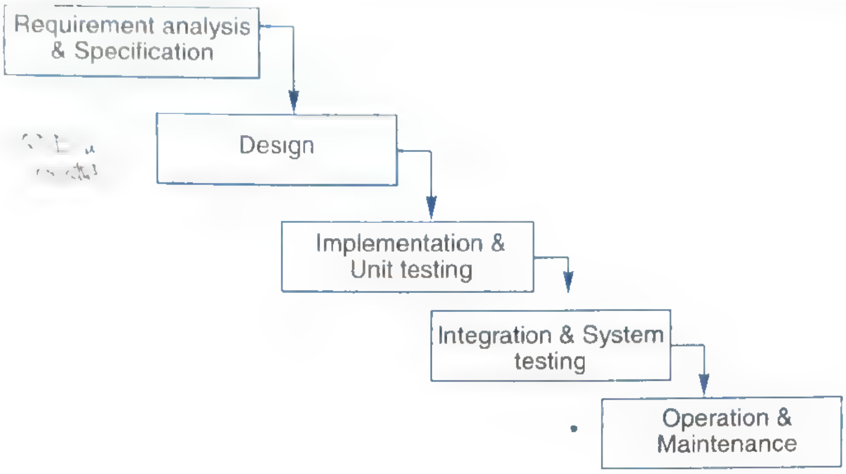


Fig. 2.2: Waterfall model

2. **Design phase:** The SRS document is produced in the previous phase, which contains the exact requirements of the customer. The goal of this phase is to transform the requirements specification into a structure that is suitable for implementation in some programming language. Here, overall software architecture is defined, and the high level and detailed design work is performed. This work is documented and known as software design description (SDD) document. The information contained in the SDD should be sufficient to begin the coding phase.

3. **Implementation and unit testing phase:** During this phase, design is implemented. If the SDD is complete, the implementation or coding phase proceeds smoothly, because all the information needed by the software developers is contained in the SDD.

During testing, the major activities are centered around the examination and modification of the code. Initially, small modules are tested in isolation from the rest of the software product. There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? Such problems are solved in this phase and modules are tested after writing some overhead code.

4. **Integration and system testing phase:** This is a very important phase. Effective testing will contribute to the delivery of higher quality software products, more satisfied users, lower maintenance costs, and more accurate and reliable results. It is a very expensive activity and consumes one-third to one half of the cost of a typical development project.

As we know, the purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing is performed. System testing involves the testing of the entire system, whereas software is a part of the system. This is essential to build confidence in the developers before software is delivered to the customer or released in the market.

5. **Operation and maintenance phase:** Software maintenance is a task that every development group has to face, when the software is delivered to the customer's site, installed and is operational. Therefore, release of software inaugurates the operation and maintenance phase of the life cycle. The time spent and effort required to keep the software operational

after release is very significant. Despite the fact that it is a very important and challenging task; it is routinely the poorly managed headache that nobody wants to face.

Software maintenance is a very broad activity that includes error correction, enhancement of capabilities, deletion of obsolete capabilities, and optimization. The purpose of this phase is to preserve the value of the software over time. This phase may span for 5 to 50 years whereas development may be 1 to 3 years.

This model is easy to understand and reinforces the notion of "define before design" and "design before code". This model expects complete and accurate requirements early in the process, which is unrealistic. Working software is not available until relatively late in the process, thus delaying the discovery of serious errors. It also does not incorporate any kind of risk assessment.

Problems of waterfall model

(*i*) It is difficult to define all requirements at the beginning of a project.

(*ii*) This model is not suitable for accomodating any change.

(*iii*) A working version of the system is not seen until late in the project's life.

(*iv*) It does not scale up well to large projects.

(*v*) Real projects are rarely sequential.

Due to these weaknesses, the application of waterfall model should be limited to situations where the requirements and their implementation are well understood. For example, if an organisation has experience in developing accounting systems then building a new accounting system based on existing designs could be easily managed with the waterfall model.

2.3 INCREMENT PROCESS MODELS

Increment process models are effective in the situations where requirements are defined precisely and there is no confusion about the functionality of the final product. (Although, functionality can be delivered in phases as per desired priorities). After every cycle, a useable product is given to the customer. For example, in the university automation software library automation module may be delivered in the first phase and examination automation module in the second phase and as so on. Every new cycle will have an additional functionality. Increment process models are popular particularly when we have to quickly deliver a limited functionality system.

2.3.1 Iterative Enhancement Model

This model has the same phases as the waterfall model, but with fewer restrictions. Generally the phases occur in the same order as in the waterfall model, but these may be conducted in several cycles. A useable product is released at the end of the each cycle, with each release providing additional functionality [BASI75].

During the first requirements analysis phase, customers and developers specify as many requirements as possible and prepare a SRS document. Developers and customers then prioritize these requirements. Developers implement the specified requirements in one or more cycles of design, implementation and test based on the defined priorities. The model is given in Fig. 2.3.

The aim of the waterfall and prototyping models is the delivery of a complete, operational and good quality product. In contrast, this model does deliver an operational quality product at each release, but one that satisfies only a subset of the customer's requirements. The complete product is divided into releases, and the developer delivers the product release by release. A typical product will usually have many releases as shown in Fig. 2.3. At each release, customer has an operational quality product that does a portion of what is required. The customer is able to do some useful work after first release. With this model, first release may be available within few weeks or months, whereas the customer generally waits months or years to receive a product using the waterfall and prototyping model.

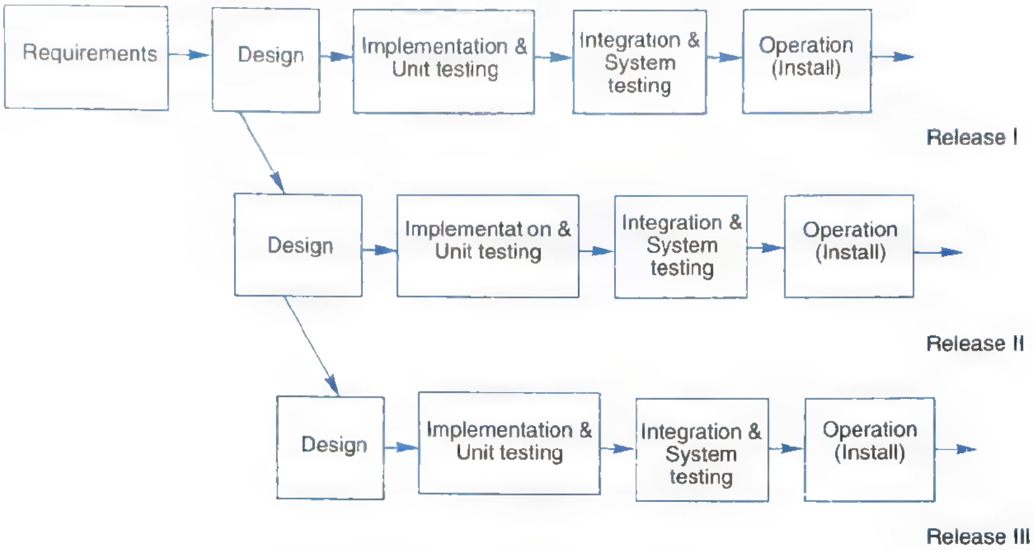


Fig. 2.3: Iterative enhancement model

2.3.2 The Rapid Application Development (RAD) Model

This model is an incremental process model and was developed by IBM in the 1980s and described in the book of James Martin entitled "Rapid Application Development". Here, user involvement is essential from requirement phase to delivery of the product. The continuous user participation ensures the involvement of user's expectations and perspective in requirements elicitation, analysis and design of the system.

The process is started with building a rapid prototype and is given to user for evaluation. The user feedback is obtained and prototype is refined. The process continues, till the requirements are finalised. We may use any grouping technique (like FAST, QFD, Brainstorming Sessions; for details refer chapter 3) for requirements elicitation. Software requirement and specification (SRS) and design documents are prepared with the association of users.

There are four phases in this model and these are shown in Fig. 2.4.

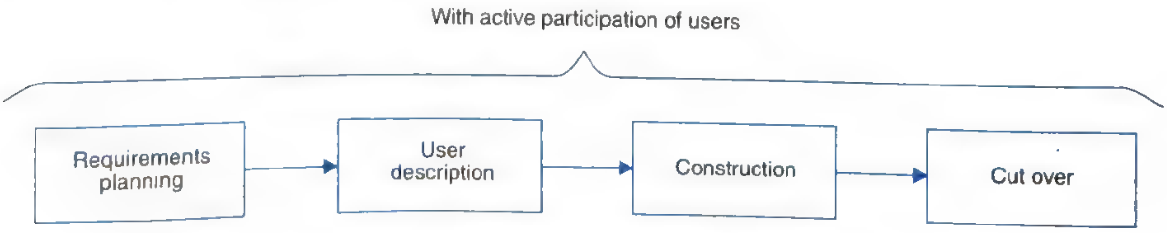


Fig. 2.4: RAD model

(*i*) **Requirements planning phase:** Requirements are captured using any group elicitation technique. Some techniques are discussed in chapter 3. Only issue is the active involvement of users for understanding the project.

(*ii*) **User description:** Joint teams of developers and users are constituted to prepare, understand and review the requirements. The team may use automated tools to capture information from the other users.

(*iii*) **Construction phase:** This phase combines the detailed design, coding and testing phase of waterfall model. Here, we release the product to customer. It is expected to use code generators, screen generators and other types of productivity tools.

(*iv*) **Cut over phase:** This phase incorporates acceptance testing by the users, installation of the system, and user training.

In this model, quick initial views about the product are possible due to delivery of rapid prototype. The development time of the product may be reduced due to use of powerful development tools. It may use CASE tools and frameworks to increase productivity. Involvement of user may increase the acceptability of the product.

If user cannot be involved throughout the life cycle, this may not be an appropriate model. Development time may not be reduced very significantly, if reusable components are not available. Highly specialized and skilled developers are expected and such developers may not be available very easily. It may not be effective, if system can not be properly modularised.

2.4 EVOLUTIONARY PROCESS MODELS

Evolutionary process model resembles iterative enhancement model. The same phases as defined for the waterfall model occur here in a cyclical fashion. This model differs from iterative enhancement model in the sense that this does not require a useable product at the end of each cycle. In evolutionary development, requirements are implemented by category rather than by priority.

For example, in a simple database application, one cycle might implement the graphical user interface (GUI); another file manipulation; another queries; and another updates. All four cycles must complete before there is working product available. GUI allows the users to interact with the system; file manipulation allows data to be saved and retrieved; queries allow users to get data out of the system; and updates allow users to put data into the system. With any one of those parts missing, the system would be unusable.

In contrast, an iterative enhancement model would start by developing a very simplistic, but usable database. On the completion of each cycle, the system would become more sophisticated. It, would, however, provide all the critical functionality by the end of the first cycle. Evolutionary development and iterative enhancement are somewhat interchangeable. Evolutionary development should be used when it is not necessary to provide a minimal version of the system quickly.

These models are useful for projects using new technology that is not well understood. This is also used for complex projects where all functionality must be delivered at one time, but the requirements are unstable or not well understood at the beginning.

2.4.1 Prototyping Model

A disadvantage of waterfall model as discussed in the last section is that the working software is not available until late in the process, thus delaying the discovery of serious errors. An alternative to this is to first develop a working prototype of the software instead of developing the actual software. The working prototype is developed as per current available requirements. Basically, it has limited functional capabilities, low reliability, and untested performance (usually low).

The developers use this prototype to refine the requirements and prepare the final specification document. Because the working prototype has been evaluated by the customer, it is reasonable to expect that the resulting specification document will be correct. When the prototype is created, it is reviewed by the customer. Typically this review gives feedback to the developers that helps to remove uncertainties in the requirements of the software, and starts an iteration of refinement in order to further clarify requirements as shown in Fig. 2.5.

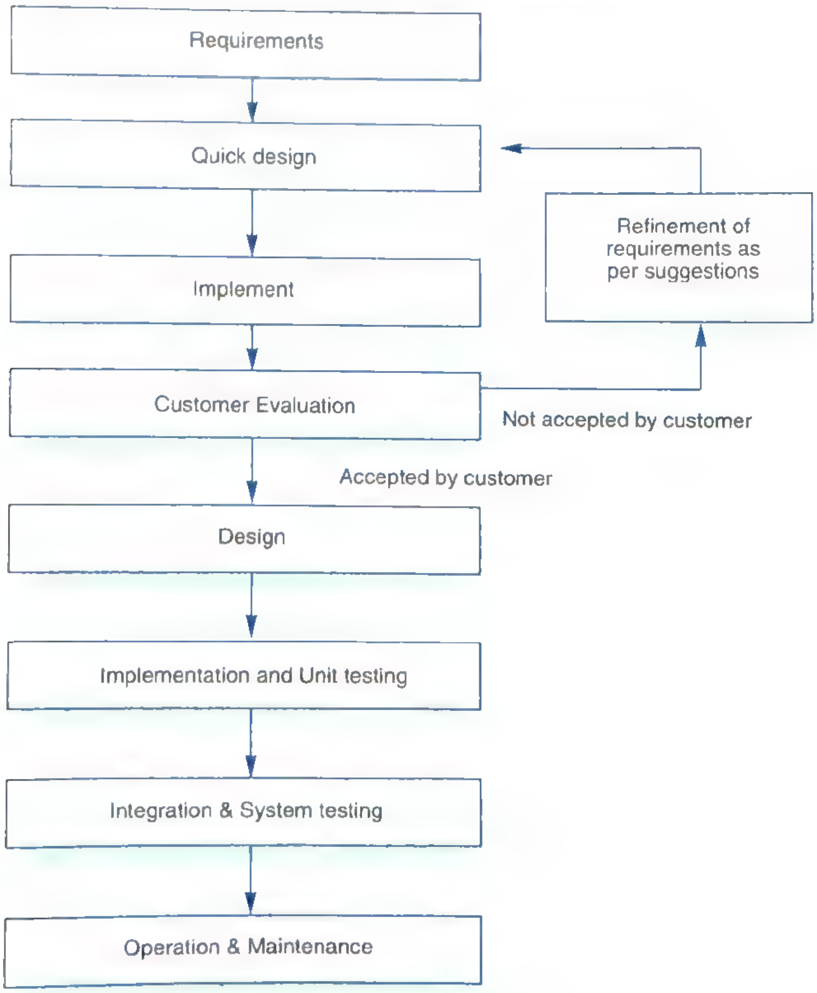


Fig. 2.5: Prototyping model

The prototype may be a usable program, but is not suitable as the final software product. The reason may be poor performance, maintainability or overall quality. The code for the prototype is thrown away; however the experience gathered from developing the prototype helps in developing the actual system. Therefore, the development of a prototype might involve

extra cost, but overall cost might turnout to be lower than that of an equivalent system developed using the waterfall model.

The developers should develop prototype as early as possible to speed up the software development process. After all, the sole use of this is to determine the customer's real needs. Once this has been determined, the prototype is discarded. For this reason, the internal structure of the prototype is not very important [SCHA96].

After the finalization of software requirement and specification (SRS) document, the prototype is discarded and actual system is then developed using the waterfall approach. Thus, it is used as an input to waterfall model and produces maintainable and good quality software. This model requires extensive participation and involvement of the customer, which is not always possible.

2.4.2 Spiral Model

The problem with traditional software process models is that they do not deal sufficiently with the uncertainty, which is inherent to software projects. Important software projects have failed because project risks were neglected and nobody was prepared when something unforeseen happened. Barry Boehm recognized this and tried to incorporate the "project risk" factor into a life cycle model. The result is the spiral model, which was presented in 1986 [BOEH86] and is shown in Fig. 2.6.

The radial dimension of the model represents the cumulative costs. Each path around the spiral is indicative of increased costs. The angular dimension represents the progress made in completing each cycle. Each loop of the spiral from X-axis clockwise through 360° represents one phase. One phase is split roughly into four sectors of major activities:

- *Planning:* Determination of objectives, alternatives and constraints
- *Risk analysis:* Analyze alternatives and attempts to identify and resolve the risks involved
- *Development:* Product development and testing product
- *Assessment:* Customer evaluation

During the first phase, planning is performed, risks are analyzed, prototypes are built, and customers evaluate the prototype. During the second phase, a more refined prototype is built, requirements are documented and validated, and customers are involved in assessing the new prototype. By the time third phase begins, risks are known, and a somewhat more traditional development approach is taken [RAKI97].

The focus is the identification of problems and the classification of these into different levels of risks, the aim being to eliminate high-risk problems before they threaten the software operation or cost.

An important feature of the spiral model is that each phase is completed with a review by the people concerned with the project (designers and programmers). This review consists of a review of all the products developed up to that point and includes the plans for the next cycle. These plans may include a partition of the product in smaller portions for development or components that are implemented by individual groups or persons. If the plan for the development fails, then the spiral is terminated. Otherwise, it terminates with the initiation of new or modified software.

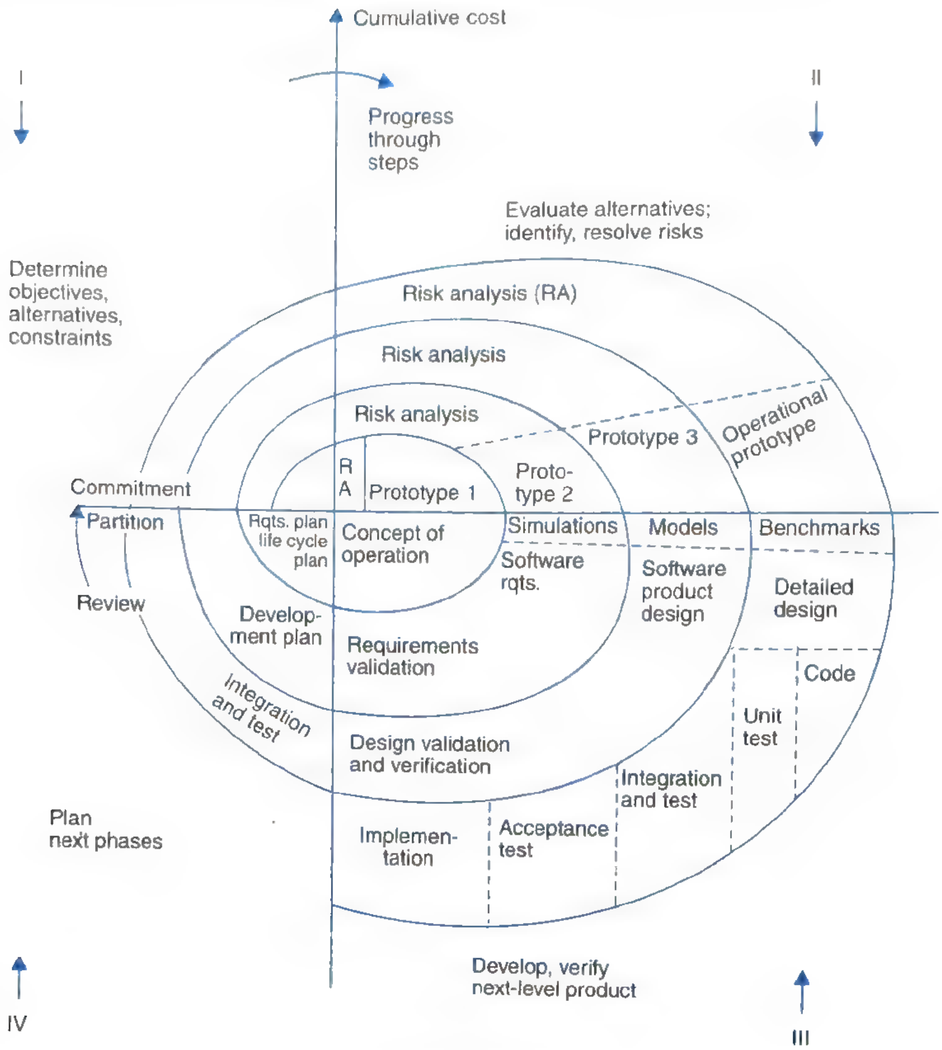


Fig. 2.6: Spiral model

The advantage of this model is the wide range of options to accommodate the good features of other life cycle models. It becomes equivalent to another life cycle model in appropriate situations. It also incorporates software quality objectives into software development. The risk analysis and validation steps eliminate errors in the early phases of development.

The spiral model has some difficulties that need to be resolved before it can be a universally applied life cycle model. These difficulties include lack of explicit process guidance in determining objectives, constraints, alternatives; relying on risk assessment expertise; and providing more flexibility than required for many applications.

2.5 THE UNIFIED PROCESS

The unified process is described by I. Jacobson, G. Booch and J. Rumbaugh in their book "The unified software development process" [JACO 98]. It is a software engineering process with the goal of producing good quality maintainable software within specified time and budget. The unified process supports iterative development where project is developed through a series of short, fixed length miniprojects called iterations. The outcome of each iteration is a tested, integrated and executable system. Each iteration has its own requirements analysis, design,

implementation and testing activities. Hence, the system continues to enlarge and refine with every iteration and thus grows incrementally over time. Thus, this approach is also known as iterative and incremental development. The unified process is now maintained and enhanced by Rational software corporation and thus also referred as Rational unified process (RUP).

As we all know, sequential approach (example waterfall model) is not suitable due to changing requirements and uncertainities in software development. Iterative approach is definitely better than sequential approach and became the basis of unified process. The process also defines who is doing what, how and when. The role of everyone in the development process is defined clearly and precisely.

2.5.1 The Phases of the Unified Process

There are four phases in the unified process and the life cycle is shown in the Figure 2.7. Each phase has a specific outcome which can be reviewed, if required, to improve the quality of the process.

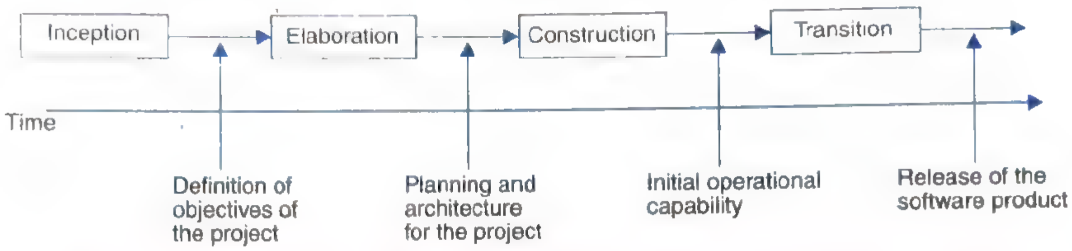


Fig. 2.7: The phases of unified process alongwith outcomes after each phase

(*i*) **Inception:** This phase is used to define the scope of the work. This is similar to requirements analysis and specification phase of waterfall model. We define the requirements using the requirements elicitation techniques. The customer and developer interaction helps us to capture and define the requirements. Although, requirements are always changing but these change can be accomodated due to iterative nature of the process. The outcome of the phase is clear definition of the objectives of the project.

(*ii*) **Elaboration:** How do we plan and design the project ? What resources are required? What type of architecture may be suitable ? These questions are answered in this phase. We specify the features and prepare the baseline of the architecture. This is similar to design phase of waterfall model. The outcome is the planning and architecture documents of the project.

(*iii*) **Construction:** The objectives are translated in design and architecture documents. These documents are the inputs to construction phase and output is the product. We build and test the product in this phase. The outcome of this phase is the deliverable product to the customer and sometimes may be treated as beta release.

(*iv*) **Transition:** Transitioning the product to the customers involves many activities like delivering, training, supporting, and maintaining the product. The ultimate objective is the customer's satisfaction. The phase activities may continue till the customers are satisfied. The outcome is the product release which also concludes the life cycle of unified process.

These four phases (Inception, Elaboration, Construction and Transition) constitute a development cycle and produce a software generation. In the first iteration *i.e.*, initial development cycle, first version of the software is produced. The software will be used by users. The

feedback of the users in terms of additional functionality, failure reports etc. may motivate us to work for second version. The same process through inception, elaboration, construction and transition phases will repeat to evolve the next generation of the software product. These cycles are known as evolution cycles as shown in figure 2.8. With several evolution cycles, new generations of the product are produced. This process will continue upto the retirement of the software product.

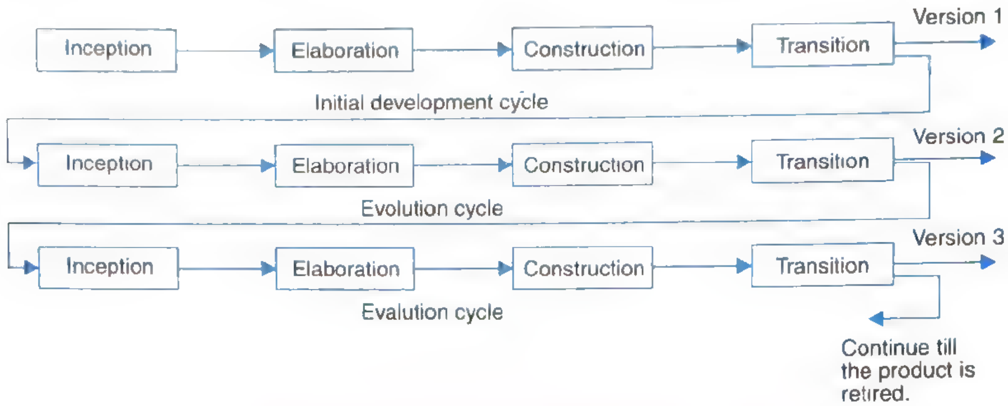


Fig. 2.8: Initial development and evolution cycles

The phases are not of equal lengths. The length may vary depending on the type, specifications and environment of the project. In each phase, we progress iteratively and each phase may consist of one or several iterations as shown in Figure 2.9.

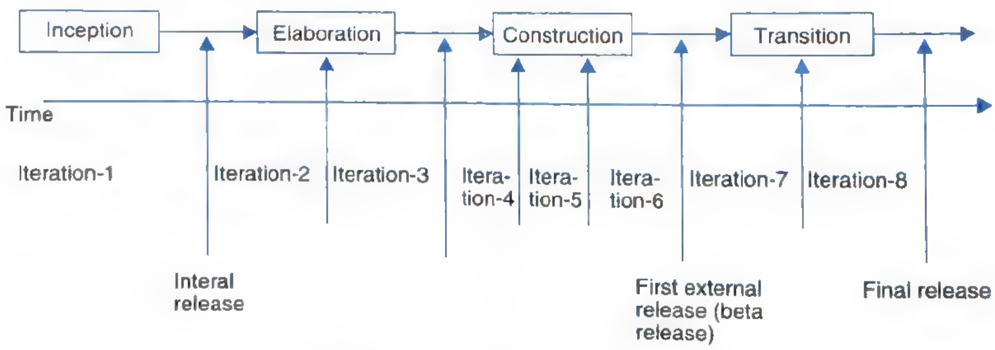


Fig. 2.9: Iterations within a phase

Each iteration follows a pattern which is similar to waterfall model. Hence, the work-flow contains the activities of requirements elicitation and analysis, design, implementation and testing. However, from one iteration to the next and from one phase to the next, the emphasis on various activities will change. Figure 2.10 shows the relative emphasis of the various types of activities over time [KRUC 99]. These activities are not separated in time. Rather, they are executed concurrently throughout the lifetime of the project. As we have seen in figure 2.10, not much code is written early, in the project life cycle ; but the amount is not zero. Late in the projected, most of the requirements are known, but some new ones are still identified. Thus, as the project matures, the emphasis on certain activities increases or decreases, but activities are allowed to execute at any time throughout the lifetime of the project [BOOC 98].

The inception phase is primarily dedicated to the requirements elicitation and analysis. The scope of the work is defined and some planning work is also carried out. The elaboration phase has the focus on requirements with some emphasis on design and implementation. During the construction phase, focus is mostly on design and implementation. We produce first operational product after this phase.

The focus of transition phase is on training, installation and customer interaction. The feedback of customer helps us to improve and enhance the product. We produce and deliver the final product here.

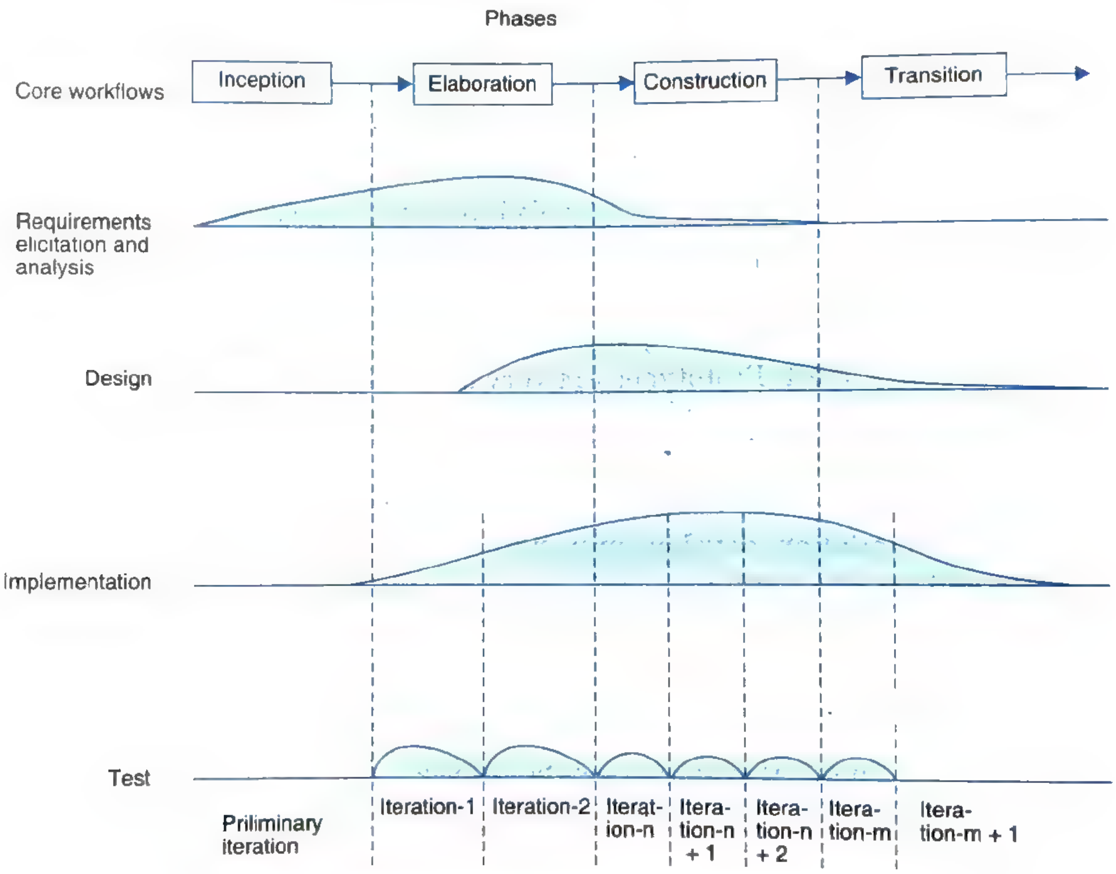


Fig. 2.10: Iterations and workflow of unified process

2.5.2 Unified Process Work Products

All four phases produce different work products as per outcome of the phases. The inception phase has the following objectives :

- Gathering and analysing the requirements.
- Planning and preparing a business case and evaluating alternatives for risk management, scheduling resources, etc.
- Estimating the overall cost and schedule for the project.
- Studing the feasibility and calculating profitability of the project.

Many documents are prepared and are shown in Fig. 2.11.

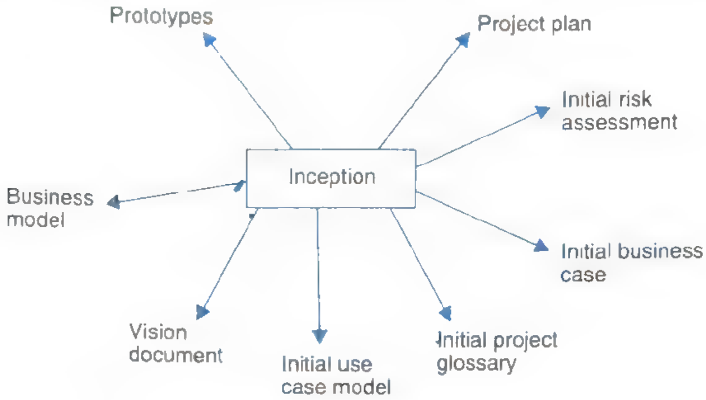


Fig. 2.11: The outcomes of Inception phase

The vision document's emphasis is on requirements and scope of the project. It gives us an idea about overall objective and expectations from the project. Initial use case model gives use case diagram alongwith actors etc. It may also consist of use cases that can be identified at this early stage.

Initial business case may include revenue model, market conditions, business difficulties and financial forecast etc. Project plan document may include phases and iterations for the development of the project. Prototypes are very useful for the understanding of the project (optional item). A business model may be required to provide idea about "time to market". marketing strategies, cash flows and threats.

The elaboration phase has the focus on the analysis of problem domain, establishes a sound architectural foundation, develops project plans and eliminates the risk elements. Some of the objectives and activities are :

- Establishing architectural foundations
- Design of use case model
- Elaborating the process, infrastructure and development environment
- Selecting component, if possible, for the project
- Demonstrating that architecture supports the vision at reasonable cost and with specified time.

The outcomes are given in Fig. 2.12.

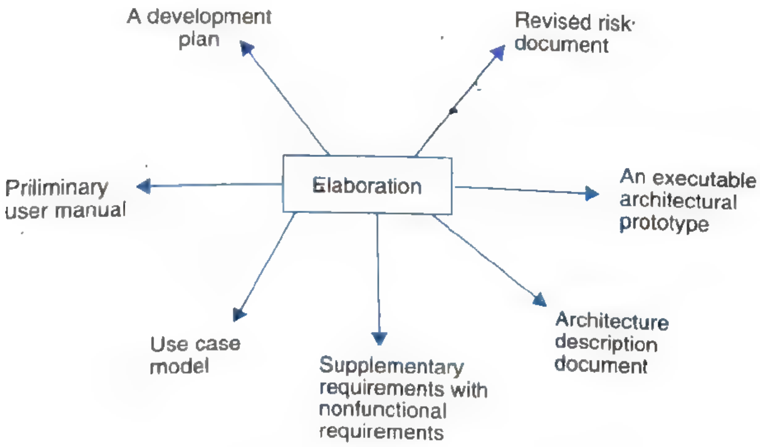


Fig. 2.12: The outcomes of elaboration phase

A use case model (at least 80% complete) is prepared. All use cases are identified alongwith associated actors. Supplementary requirements document (with nonfunctional requirements) and software architecture description document are also prepared. A prototype is also designed for the customer to give an idea about the final product. A development plan with iterations, workflows etc. is also prepared. A preliminary user m nual, although optional, may also be prepared in this phase and will refine in the subsequent phases.

The construction phase produces the product for the customer. The objectives and activities are :

- Implementing the project
- Minimising development cost
- Managing and optimizing resources
- Testing the product
- Assessing the product releases against acceptance criteria.

The outcome of this phase is a product ready to use for the customers and is shown in Fig. 2.13.

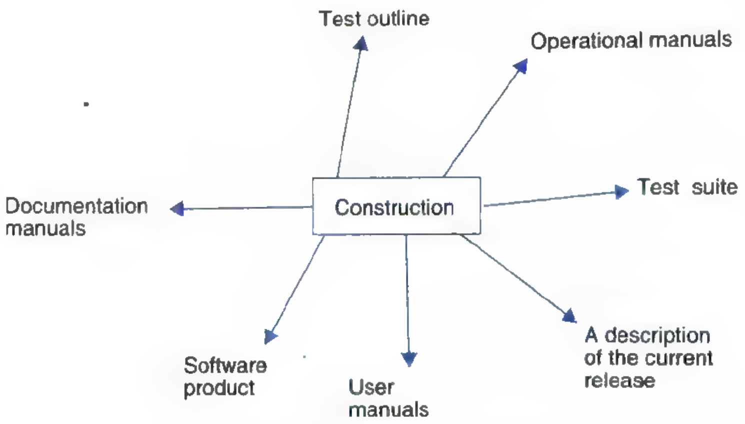


Fig. 2.13: The outcomes of construction phase

The outcomes are various operational and documentation manuals alongwith software product. The test suite documents are also preserved carefully because they are required during the maintenance of the software product.

The transition phase lays focus on successful delivery and installation of the software product. The phase includes the following :

- Commencement of beta testing
- Analysis of user's views
- Training of users
- Tuning activities including bug fixing and enhancement for performance and usability
- Assessing the customer satisfaction

The outcomes are given in Fig. 2.14.

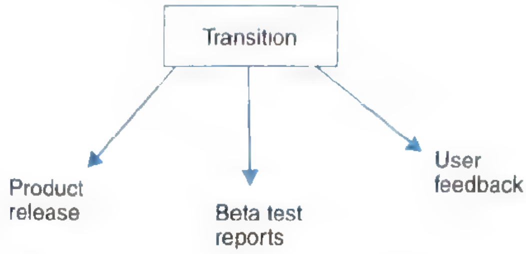


Fig. 2.14: The outcomes of transition phase

The unified process has many advantages due to its iterative and incremental nature. Changes can be accomodated, risks can be minimised, reusability can be ensured and learning alongwith project evolution is possible. Hence, the product that results from unified process will be of good quality. The system has been tested several times, improving the quality of testing. The requirements have been refined and are more closely related to the customer's actual expectations.

2.6 SELECTION OF A LIFE CYCLE MODEL

The selection of a suitable model is based on the following characteristics/categories:

(*i*) Requirements

(*ii*) Development team

(*iii*) Users

(*iv*) Project type and associated risk.

2.6.1 Characteristics of Requirements

Requirements are very important for the selection of an appropriate model. There are number of situations and problems during requirements capturing and analysis. The details are given in Table 2.1.

Table 2.1: Selection of a model based on characteristics of requirements

Requirements	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
Are requirements easily understandable and defined?	Yes	No	No	No	No	Yes
Do we change requirements quite often?	No	Yes	No	No	Yes	No
Can we define requirements early in the cycle?	Yes	No	Yes	Yes	No	Yes
Requirements are indicating a complex system to be built	No	Yes	Yes	Yes	Yes	No

2.6.2 Status of Development Team

The status of development team in terms of availability, effectiveness, knowledge, intelligence, team work etc., is very important for the success of the project. If we know above mentioned parameters and characteristics of the team, then we may choose an appropriate life cycle model for the project. Some of the details are given in Table 2.2.

Table 2.2: Selection based on status of development team

Development team	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
Less experience on similar projects	No	Yes	No	No	Yes	No
Less domain knowledge (new to the technology)	Yes	No	Yes	Yes	Yes	No
Less experience on tools to be used	Yes	No	No	No	Yes	No
Availability of training, if required	No	No	Yes	Yes	No	Yes

2.6.3 Involvement of Users

Involvement of users increases the understandability of the project. Hence user participation, if available, plays a very significant role in the selection of an appropriate life cycle model. Some issues are discussed in Table 2.3.

Table 2.3: Selection based on user's participation

Involvement of users	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
User involvement in all phases	No	Yes	No	No	No	Yes
Limited user participation	Yes	No	Yes	Yes	Yes	No
User have no previous experience of participation in similar projects	No	Yes	Yes	Yes	Yes	No
Users are experts of problem domain	No	Yes	Yes	Yes	No	Yes

2.6.4 Type of Project and Associated Risk

Very few models incorporate risk assessment. Project type is also important for the selection of a model. Some issues are discussed in Table 2.4.

Table 2.4: Selection based on type of project with associated risk

Project type and risk	*Waterfall*	*Prototype*	*Iterative enhancement*	*Evolutionary development*	*Spiral*	*RAD*
Project is the enhancement of the existing system	No	No	Yes	Yes	No	Yes
Funding is stable for the project	Yes	Yes	No	No	No	Yes
High reliability requirements	No	No	Yes	Yes	Yes	No
Tight project schedule	No	Yes	Yes	Yes	Yes	Yes
Use of reusable components	No	Yes	No	No	Yes	Yes
Are resources (time, money people etc.) scarce?	No	Yes	No	No	Yes	No

An appropriate model may be selected based on options given in four Tables (*i.e.*, Table 2.1 to 2.4). Firstly, we have to answer the questions presented for each category by circling a yes or no in each table. Rank the importance of each category, or question within the category, in terms of the project for which we want to select a model. The total number of circled responses for each column in the tables decide an appropriate model. We may also use the category ranking to resolve the conflicts between models if the total in either case is close or the same.

REFERENCES

[BASI75] Basili V.R., "*Iterative Enhancement: A Practical Technique for Software Development*", IEEE Trans on Software Engineering, SE–1, No. 4, 390–396, December 1975.

[BOEH86] Boehm B., "*A Spiral Model for Software Development and Enhancement*", ACM Software Engineering Notes, 14–24, August, 1986.

[BOOC98] Booch G. et. al., "*Object Oriented Analysis and Design with Applications*", 2nd ed., Addison Wesley Longman Inc., 1998.

[JACO98] Jacobson I., Booch G. and Rumbaugh J., "*The Unified Software Development Process*", Reading, MA : Addison Wesley Longmen, 1998.

[KRUC99] Kruchten P., "*The Rational Unified Process—An Introduction*", Addison Wesley Longman Inc. 1999.

[RAKI97] Rakitin S.R., "*Software Verification and Validation*", Artech House Inc., Norwood, MA, 1997.

[SCHA96] Schach S., "*Classical and Object Oriented Software Engineering*", IRWIN, USA, 1996.

[TAKA96] Takang A.A. and Grubb P.A., "*Software Maintenance–Concepts and Practice*", Int. Thomson Computer Press, Cambridge, U.K., 1996.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions.

2.1. Spiral Model was developed by
(*a*) Bev Littlewood (*b*) Berry Boehm
(*c*) Roger Pressman (*d*) Victor Basili.

2.2. Which model is most popular for student's small projects?
(*a*) waterfall model (*b*) spiral model
(*c*) quick and fix model (*d*) prototyping model.

2.3. Which is not a software life cycle model?
(*a*) waterfall model (*b*) spiral model
(*c*) prototyping model (*d*) capability maturity model.

2.4. Project risk factor is considered in
(*a*) waterfall model (*b*) prototyping model
(*c*) spiral model (*d*) iterative enhancement model.

2.5. SDLC stands for
(*a*) software design life cycle (*b*) software development life cycle
(*c*) system development life cycle (*d*) system design life cycle.

2.6. Build and fix model has
(*a*) 3 phases (*b*) 1 phase
(*c*) 2 phases (*d*) 4 phases.

2.7. SRS Stands for
(*a*) software requirements specification (*b*) software requirements solutions
(*c*) system requirements specification (*d*) none of the above.

2.8. Waterfall model is not suitable for
(*a*) small projects (*b*) accomodating change
(*c*) complex projects (*d*) none of the above.

2.9. RAD stands for
(*a*) rapid application development (*b*) relative application development
(*c*) ready application development (*d*) repeated application development.

2.10. RAD model was proposed by
(*a*) Lucent Technologies (*b*) motorola
(*c*) IBM (*d*) microsoft.

2.11. If requirements are easily understandable and defined, which model is best suited?
(*a*) waterfall model (*b*) prototyping model
(*c*) spiral model (*d*) none of the above.

2.12. If requirements are frequently changing, which model is to be selected
(*a*) waterfall (*b*) prototyping model
(*c*) RAD model (*d*) iterative enhancement model.

2.13. If user participation is available, which model is to be chosen?
(*a*) waterfall model (*b*) iterative enhancement model
(*c*) spiral model (*d*) RAD model.

2.14. If limited user participation is available, which model is to be selected?
(a) waterfall model (b) spiral model
(c) iterative enhancement model (d) any of the above.

2.15. If project is the enhancement of existing system, which model is best suited?
(a) waterfall model (b) prototyping model
(c) iterative enhancement model (d) spiral model.

2.16. Which one is the most important feature of spiral model?
(a) quality management (b) risk management
(c) performance management (d) efficiency management.

2.17. Most suitable model for new technology that is not well understood is:
(a) waterfall model (b) RAD model
(c) iterative enhancement model (d) evolutionary development model.

2.18. Statistically, the maximum percentage of errors belong to the following phase of SDLC
(a) coding (b) design
(c) specifications (d) installation and maintenance.

2.19. Which phase is not available in software life cycle?
(a) coding (b) testing
(c) maintenance (d) abstraction.

2.20. The development is supposed to proceed linearly through the phases in
(a) spiral model (b) waterfall model
(c) prototyping model (d) none of the above.

2.21. Unified Process is maintained by
(a) infosys (b) rational software corporation
(c) SUN Microsystem (d) none of the above.

2.22. Unified Process is
(a) interative (b) incremental
(c) evolutionary (d) all of the above.

2.23. Who is not in the team of unified process development ?
(a) I. Jacobson (b) G. Booch.
(c) B. Boehm (d) J. Rumbaugh.

2.24. How many phases are in the unified process ?
(a) 4 (b) 5
(c) 3 (d) none of the above.

2.25. The outcome of construction phase can be treated as :
(a) product release (b) beta release
(c) alpha release (d) all of the above.

EXERCISE

2.1. What do you understand by the term Software Development Life Cycle (SDLC)? Why is it important to adhere to a life cycle model while developing a large software product?

2.2. What is software life cycle? Discuss the generic waterfall model.

2.3. List the advantages of using waterfall model instead of adhoc build and fix model.

2.4. Discuss the prototype model. What is the effect of designing a prototype on the overall cost of the software project?

2.5. What are the advantages of developing the prototype of a system?

2.6. Describe the type of situations where iterative enhancement model might lead to difficulties.

2.7. Compare iterative enhancement model and evolutionary process model.

2.8. Sketch a neat diagram of spiral model of software life cycle.

2.9. Compare the waterfall model and the spiral model of software development.

2.10. As we move outward alongwith process flow path of the spiral model, what can we say about the software that is being developed or maintained?

2.11. How does "project risk" factor affect the spiral model of software development?

2.12. List the advantages and disadvantages of involving a software engineer throughout the software development planning process.

2.13. Explain the spiral model of software development. What are the limitations of such a model?

2.14. Describe the rapid application development (RAD) model. Discuss each phase in detail.

2.15. What are the characteristics to be considered for the selection of a life cycle model?

2.16. What is the role of user participation in the selection of a life cycle model?

2.17. Why do we feel that characteristics of requirements play a very significant role in the selection of a life cycle model?

2.18. Write short note on "status of development team" for the selection of a life cycle model.

2.19. Discuss the selection process parameters for a life cycle model.

2.20. What is unified process? Explain various phases alongwith outcome of each phase.

2.21. Describe the unified process work products after each phase of unified process.

2.22. What are the advantages of iterative approach over sequential approach? Why is unified process called as iterative and incremental?

3 Software Requirements : Analysis and Specifications

When we receive a request for a new software project from the customer, first of all, we would like to understand the project. The new project may replace the existing system such as preparation of students semester results electronically rather than manually. Sometimes, the new project is an enhancement or extension of a current (manual or automated) system. For example, a web enabled student result declaration system that would enhance the capabilities of the current result declaration system. No matter, whether its functionality is old or new, each project has a purpose, usually expressed in what the system can do. Hence, goal is to understand the requirements of the customer and document them properly. A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfil the system's purpose.

The hardest part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later [BROO95]. Throughout software industry's history, we have struggled with this truth. Defining and applying good, complete requirements is hard to work, and success in this endeavor has eluded many of us. Yet, we continue to make progress.

3.1 REQUIREMENTS ENGINEERING

Requirements describe the "what" of a system, not the "how". Requirements engineering produces one large document, written in a natural language, contains a description of what the system will do without describing how it will do. The input to requirements engineering is the problem statement prepared by the customer. The problem statement may give an overview of the existing system alongwith broad expections from the new system.

3.1.1 Crucial Process Steps

The quality of a software product is only as good as the process that creates it. Requirements engineering is one of the most crucial activity in this creation process. Without well-written requirements specifications, developers do not know what to build, customers do not know what to expect, and there is no way to validate that the built system satisfies the requirements.

Requirements engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behaviour and its associated constraints [HSIA93]. This process consists of four steps as shown in Fig. 3.1.

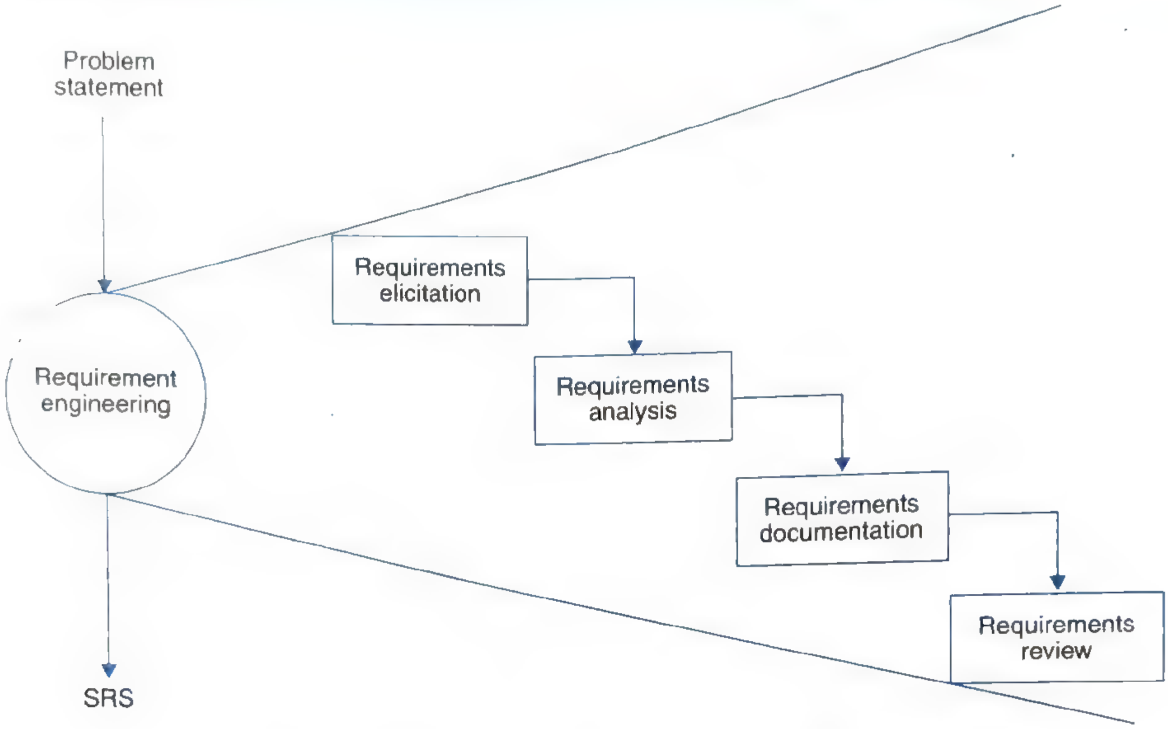


Fig. 3.1: Crucial process steps of requirement engineering

(*i*) **Requirements elicitation:** This is also known as gathering of requirements. Here, requirements are identified with the help of customer and existing systems processes, if available.

(*ii*) **Requirements analysis:** Analysis of requirements starts with requirement elicitation. The requirements are analysed in order to identify inconsistencies, defects, omissions etc. We describe requirements in terms of relationships and also resolve conflicts, if any.

(*iii*) **Requirements documentation:** This is the end product of requirements elicitation and analysis. The documentation is very important as it will be the foundation for the design of the software. The document is known as software requirements specification (SRS).

(*iv*) **Requirements review:** The review process is carried out to improve the quality of the SRS. It may also be called as requirements verification. For maximum benefits, review and verification should not be treated as a discrete activity to be done only at the end of the preparation of SRS. It should be treated as continuous activity that is incorporated into the elicitation, analysis, and documentation.

The primary output of requirements engineering is requirements specifications. If it describes both hardware and software, it is a system requirements specification. If it describes only software, it is a software requirements specification. In either case, a requirements specification must treat the system as a black box. It must delineate inputs, outputs, the functional requirements that show external behaviour in terms of input, output, and their relationships, and non-functional requirements and their constraints, including performance, portability, and reliability.

The software requirements specification (SRS) should be internally consistent; consistent with existing documents; correct and complete with respect to satisfying needs; understandable

to users, customers, designers, and testers; and capable of serving as a basis for both design and test. This SRS document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure in implementing the contracted system.

3.1.2 Present State of Practice

Most software development organizations agree to the fact that there should be a set of activities called requirements engineering and their success is vital to the success of the entire project. So why is the state of the practice no better than it is? There are several reasons, not all of them obvious [BERR98, DAV194, HSIA93]; and some are discussed below:

1. **Requirements are difficult to uncover:** Today we are automating virtually every kind of task-some that were previously done manually and some that have never been done before. In either kind of application, it is difficult, if not impossible to identify all the requirements, regardless of the techniques we use. No one can see a brand new system in its entirety. Even if someone could, the description is always incomplete at start. Users and developers must resort to trial and error to identify problems and solutions.

2. **Requirements change:** Because no user can come up with a complete list of requirements at the outset, the requirements get added and changed as the user begins to understand the system and his or her real needs. That is why we always have requirement changes. But, project schedule is seldom adjusted to reflect these modifications. Fluid requirements make it difficult to establish a baseline from which to design and test. Finally, it is hard to justify spending resources to make a requirement specification "perfect", because it will soon change anyway. This is the biggest problem, and there is as yet no technology to overcome it. This problem is often used as an excuse to either eliminate or scale back requirements engineering effort.

3. **Over-reliance on CASE tools:** Computer Aided Software Engineering (CASE) tools are often sold as panaceas. These exaggerated claims, have created a false sense of trust, which could inflict untold damage on the Software Industry. CASE tools are as important to developers (including requirement writers) as word processors are to authors. However, we must not rely on requirements engineering tools without first understanding and establishing requirements engineering principles, techniques and processes. Furthermore, we must have realistic expectations from the tools.

4. **Tight project schedule:** Because of either lack of planning or unreasonable customer demand, many projects start with insufficient time to do a decent job. Sometimes, even the allocated time is reduced while the project is under way. It is also customary to reduce time set apart to analyze requirements, for early start of designing and coding, which frequently leads to disaster.

5. **Communication barriers:** Requirement engineering is communication intensive activity. Users and developers have different vocabularies, professional backgrounds, and tastes. Developers usually want more precise specifications while users prefer natural language. Selecting either results in misunderstanding and confusion.

6. **Market-driven software development:** Many of the software development is today market driven, developed to satisfy anonymous customers and to keep them coming back to buy upgrades.

7. **Lack of resources:** There may not be enough resources to build software that can do everything the customer wants. It is essential to rank requirements so that, in the face of pressure to release the software quickly, the most important can be implemented first.

Requirement problems are expensive and plague almost all systems and software development organizations. In most cases, the best we can hope for it is to detect errors in the requirements in time to contain them before the software is released [SAWY99]. Because of the concern for public safety, reputation and capital investments; developers began to recognize the need for clear, concise and complete requirements [COUN99].

Example 3.1: *A university wishes to develop a software system for the student result management of its M. Tech. Programme. A problem statement is to be prepared for the software development company. The problem statement may give an overview of the existing system and broad expectation from the new software system.*

Solution: The problem statement is prepared by the Examination division of the University and is given below:

"A University conducts a 4-semester M. Tech programme. The students are offered four theory papers and two Lab papers (practicals) during Ist, IInd and IIIrd semesters. The theory papers offered in these semesters are categorized as either 'Core' or 'Elective'. Core papers do not have an alternative subject, whereas elective papers may have two or more alternative subjects. Thus a student can study any subject out of the choices available for an elective paper.

In Ist, IInd and IIIrd semesters, 2 core papers and 2 elective papers are offered to each student. The students are also required to submit a term paper minor project in IInd and IIIrd semesters each. In IVth semester the students have to give a seminar and submit a dissertation on a topic/subject area of their interest.

The evaluation of each subject is done out of 100 marks. During the semester, minor exams are conducted for each semester. Students are also required to submit assignments as directed by the corresponding faculty and maintain Lab records for practicals. Based on the students' performance in minor exams, assignments, Lab records and their attendance, marks out of 40 are given in each theory paper and practical paper. These marks out of 40 account for internal evaluation of the students. At the end of each semester, major exams are conducted in each subject (theory as well as practical). These exams are evaluated out of 60 marks and account for external evaluation of the students. Thus, the total marks of a student in a subject are obtained by adding the marks obtained in internal and external evaluation.

Every subject has some credit points assigned to it. If the total marks of a student are > = 50 in a subject, he/she is considered 'Pass' in that subject otherwise the student is considered 'Fail' in that subject. If a student passes in a subject he/she earns all the credit points assigned to that subject, but if the students fails in a subject he/she does not earn any credit point in that subject. At any time, the latest information about subjects being offered in various semesters and their credit points can be obtained from University Website.

It is required to develop a system that will manage information about subjects offered in various semesters, students enrolled in various semesters, elective (s) opted by various students in different semesters, marks and credit points obtained by students in different semesters. The system should also have the ability to generate printable mark-sheets for each student. Semester-wise detailed mark-lists and student performance reports also need to be generated.

Example 3.2: *A university wishes to develop a software system for library management activities. Design the problem statement for the software company.*

Solution: The problem statement prepared by the library staff of the university is given below. Here activities are explained point wise rather than paragraph wise.

A software has to be developed for automating the manual library system of a University. The system should be standalone in nature. It should be designed to provide functionalities as explained below:

1. Issue of books

(*a*) A student of any course should be able to get books issued.

(*b*) Books from **General section** are issued to all but **Book bank** books are issued only for their respective courses.

(*c*) A limitation is imposed on the number of books a student can be issued.

(*d*) A maximum of 4 books from book bank and 3 books from general section per student is allowed.

(*e*) The books from book bank are issued for entire semester while books from general section are issued for 15 days only.

(*f*) The software takes the current system date as the date of issue and calculates the corresponding date of return.

(*g*) A bar code detector is used to save the student information as well as book information.

(*h*) The due date for return of the book is stamped on the book.

2. Return of books

(*a*) Any person can return the issued books.

(*b*) The student information is displayed using the bar code detector.

(*c*) The system displays the student details on whose name the books were issued as well as the date of issue and return of the book.

(*d*) The system operator verifies the duration for the issue and if the book is being returned after the specified due date, a fine of Re 1 is charged for each day.

(*e*) The information is saved and the corresponding updations take place in the database.

3. Query processing

(*a*) The system should be able to provide information like:

(*i*) Availability of a particular book

(*ii*) Availability of books of any particular author.

(*iii*) Number of copies available of the desired book.

(*b*) The system should be able to reserve a book for a particular student for 24 hrs if that book is not currently available.

The system should also be able to generate reports regarding the details of the books available in the library at any given time.

The corresponding printouts for each entry (issue/return) made in the system should be generated.

Security provisions like the login authenticity should be provided. Each user should have a user id and a password. Record of the users of the system should be kept in the log file.

Provision should be made for full backup of the system.

3.2 TYPE OF REQUIREMENTS

There are different types of requirements such as:

(*i*) Known requirements—Something a stakeholder believes to be implemented.

(*ii*) Unknown requirements—Forgotten by the stakeholder because they are not needed right now or needed only by another stakeholder.

(*iii*) Undreamt requirements—Stakeholder may not be able to think of new requirements due to limited domain knowledge.

The term stakeholder is used to refer to any one who may have some direct or indirect influence on the system requirements. Stakeholder includes end-users who will interact with the system and every one else in an organisation who will be affected by it [SOMM01].

A known, unknown, or undreamt requirement may be functional or non-functional. Functional requirements describe what the software has to do. They are often called product features.

Non-functional requirements are mostly quality requirements that stipulate how well the software does what it has to do. Non-functional quality requirements that are especially important to users include specifications of desired performance, availability, reliability, usability and flexibility. Non-functional requirements for developers are maintainability, portability, and testability.

Some requirements are architectural, such as component-naming compatibility, interfaceability, upgradability, etc. Other requirements are constraints, such as system design constraints, standards conformance, legal issues and organisational issues. Constraints can come from users or organisations and may be functional or non-functional [ROBE02].

3.2.1 Functional and Non-functional Requirements

Software requirements are broadly classified as functional and non-functional requirements.

(*i*) **Functional requirements:** These are related to the expectations from the intended software. They describe what the software has to do. They are also called product features. Sometimes, functional requirements may also specifiy what the software should not do.

(*ii*) **Non-Functional requirements:** Non-functional requirements are mostly quality requirements that stipulate how well the software does what it has to do. Non-functional requirements that are especially important to users include specifications of desired performance, availability, reliability, usability and flexibility. Non-functional requirements for developers are maintainability, portability and testability.

The functional requirements are directly related to customer's expectations and are essential for the acceptance of product. However, non-functional requirements may make the customer happy and satisfied. These requirements are important for the success of any product. Sometimes, distinction between functional and non-functional may not be easy. If we want to develop a secure system, security becomes crucial to us. It is a non-functional requirements apparently, but when design is carried out, it may have serious implications. These implications may also be reflected in functional requirements and should therefore find a place in the functional requirements category of SRS document.

The term stakeholder is used to refer to anyone who may have some direct or indirect influence on the system requirements. "Stakeholder" includes end-users who will interact with the system and everyone else in an organisation who will be affected by it [SOMM01].

Some requirements are known requirements which a stakeholder believes to be implemented. Some are unknown to one stakeholder but may be required by another. Some requirements may be undreamt requirements for the stakeholder due to limited domain knowledge. However, a known, unknown, or undreamt requirement may be functional or non-functional depending upon its nature.

3.2.2 User and System Requirements

User requirements are written for the users and include functional and non-functional requirements. Users may not be the experts of the software field ; hence simple language should be used. The software terminologies, notations etc. should be avoided. User requirements should specify the external behaviour of the system with some constraints and quality parameters. However, design issues should be avoided. Hence, we should only highlight the overview of the system without design characteristics. System requirements are derived from user requirements. They are expanded form of user requirements. They may be used as input to the designers for the preparation of software design document. The user and system requirements are the parts of software requirements and specification (SRS) document. There are various ways of writing these requirements ; however IEEE std. 830—1998 (for SRS) framework is very popular and used by many software organisations.

3.2.3 Interface Specification

Interface issues are very important for the customers. A good software may not be appriciated if interfaces are not as per expectations of the customers. The types of interfaces are :

(i) **Procedural interfaces:** Some services are offered by calling interface procedures. These interfaces are called Application Programming Interfaces (APIs).

(ii) **Data structures:** Data structures are used to transfer information from one module to another module. This can be represented using graphical data models.

(iii) **Representations of data:** The issues related to ordering of bits are established here. Some are common in embedded systems and microprocessor applications. One of the methods to describe these is to use a diagram of the structure with annotations explaining the function of each group [SUMM01].

Some formal notations are available in the literature with some strengths and weaknesses.

3.3 FEASIBILITY STUDIES

Is cancellation of a project a bad news ? As per IBM report, "31% projects get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts" [IBMG2K]. Caper Jones reports that the average cancelled project in the US is about a year behind schedule and has consumed 200% of its expected budget by the time it is cancelled [CAPE94]. He further estimates that the work on cancelled projects comprises about 15% of the total US software efforts.

In spite of these statistics, cancelling a project is, in itself, neither good nor bad. Cancelling a project later than necessary is bad. The trick is to perform the minimum amount of work necessary to determine whether the project should be cancelled.

How do we cancel a project with the least work ? One of the most effective ways is to conduct a feasibility study to determine whether the full scale project is workable. This study culminates in a feasibility report, which enables project team, customer, or upper management make a go/no go decision about rest of the project. Feasibility studies are a time tested practice, but they are not used very much. One reason might be the term "feasibility study" itself. The term sometimes relates to technical feasibility and question of technical feasibility rarely enters in the minds of software developers. If we know that project is technically feasible, why should we conduct a feasibility study ?

For few projects, technical feasibility may be very important and significant. Is it technically feasible to provide direct communication connectivity through space from one location of globe to another location ? It is technically feasible to design a programming language using "Sanskrit" (Ancient Indian Language) ? However, for most of the projects, feasibility depends on non-technical issues like :

— Are the project's cost and schedule assumptions realistic ?

— Is the business model realistic ?

— Is there any market for the product ?

3.3.1 Purpose of Feasibility Studies

The decision to implement any new project or program must be based on a thorough analysis of the proposed project/program.

A feasibility study is defined as an ~~evaluation or~~ analysis of the potential impact of a proposed project ~~or program~~. It is conducted to assist decision makers in determining whether or not to implement a project/program. It is based on extensive research on both the current practices and the proposed project or program and its impact on the system as a whole. It should also contain extensive data analysis related to financial and operational impact and should include advantages and disadvantages of both the current situation and the proposed project.

3.3.2 Focus of Feasibility Studies

Steve McConnel [SIEV98] has given the following points on which a feasibility study should focus and give proper emphasis to get good results:

(*i*) Is the product concept viable ?

(*ii*) Will it be possible to develop a product that matches the project's vision statement ?

(*iii*) What are the current estimated cost and schedule for the project ?

(*iv*) How big is the gap between the original cost and schedule targets and current estimates ?

(*v*) Is the business model for software justified when the current cost and schedule estimates are considered ?

(*vi*) Have the major risks to the project been identified and can they be surmounted ?

(*vii*) Are the specifications complete and stable enough to support remaining development work ?

(*viii*) Have users and developers been able to agree on a detailed user interface prototype ? If not, are the requirements really stable ?

(*ix*) Is the software development plan complete and adequate to support further development work ?

The work done during the first 10 to 20 percent of the project should sufficiently answer these questions and give the client or top management enough information to decide whether to fund the rest of the project.

Breaking a software project into a feasibility study phase and a main development phase (starting from requirements elicitation) helps software organisations in at least three ways:

(*i*) Some people view any cancelled project as a failure. But a project cancelled at 10 to 20 percent level vis-a-vis completion point should not be considered a failure. Cancelling one project that ultimately goes no where after it is only 10 to 20 percent complete instead of 80 to 90 percent complete should always be a good decision.

(*ii*) Feasibility study can be conducted with marginal funds. After the final decision about project, more accurate and realistic estimate can be prepared. This may help us to reduce the cost variations.

(*iii*) The project team will complete 10 to 20 percent of a project before requesting funding for the rest of it forces a focus on upstream activities that are critical to a project's success. Otherwise these activities are often ignored, and the damaging consequences of such neglect would not become apparent until late in the project.

3.4 REQUIREMENTS ELICITATION

Requirements elicitation is perhaps the most difficult, most critical, most error-prone, and most communication intensive aspect of software development. Elicitation can succeed only through an effective customer-developer partnership [WIEG99].

The real requirements actually reside in user's mind. Hence the most important goal of requirement engineering is to find out what users really need. Users need can be identified only if we understand the expectations of the users from the desired software.

It is the activity that helps to understand the problem to be solved. Requirements are gathered by asking questions, writing down the answers, asking other questions, etc. Hence, requirements gathering is the most communications intensive activity of software development. Developers and Users have different mind set, expertise and vocabularies. Due to communication gap, there are chances of conflicts that may lead to inconsistencies, misunderstanding and omission of requirements.

Therefore, requirements elicitation requires the collaboration of several groups of participants who have different background. On the one hand, customers and users have a solid background in their domain and have a general idea of what the software should do. However, they may have little knowledge of solfware development processes. On the other hand, the developers have experience in developing software but may have little knowledge of everyday environment of the users. Moreover each group may be using incompatible terminologies.

There are number of requirements elicitation methods and few of them are discussed in the following sections. Some people think that one methodology is applicable to all situations, however, generally speaking, one methodology cannot possibly be sufficient for all conditions [MACA96]. We select a particular methodology for the following reason(s):

(*i*) It is the only method that we know.

(*ii*) It is our favorite method for all situations.

(*iii*) We understand intuitively that the method is effective in the present circumstances.

Clearly, third reason demonstrates the most maturity and leads to improved understanding of stakeholder's needs and thus resulting system will satisfy those needs. Unfortunately, most of us do not have the insight necessary to make such an informed decision, and therefore rely on the first two reasons [HICK03].

3.4.1 Interviews

After receiving the problem statement from the customer, the first step is to arrange a meeting with the customer. During the meeting or interview, both the parties would like to understand each other. Normally specialised developers, often called 'requirement engineers' interact with the customer. The objective of conducting an interview is to understand the customer's expectations from the software. Both parties have different feelings, goals, opinions, vocabularies, understandings, but one thing is common, both want the project to be a success. With this in mind, requirement engineers normally arrange interviews. Requirement engineers must be open minded and should not approach the interview with pre-conceived notions about what is required.

Interview may be open-ended or structured. In open-ended interview, there is no pre-set agenda. Context free questions may be asked to understand the problem and to have an overview of the situation. For example, for a "result management system", requirement engineer may ask:

- Who is the controller of examination ?
- Who has requested for such a software ?
- How many officers are placed in the examination division ?
- Who will use the software ?
- Who will explain the manual system ?

- Is there any opposition for this project ?
- How many stakeholders are computer friendly ?

Such questions help to identify all stakeholders who will have interest in the software to be developed.

In structured interview, agenda of fairly open questions is prepared. Sometimes a proper questionnaire is designed for the interview. Interview may be started with simple questions to set people at ease. After making atmosphere comfortable and calm, specific questions may be asked to understand the requirements. The customer may be allowed to voice his or her perceptions about a possible solution.

Selection of stakeholder: It will be impossible to interview every stakeholder. Thus, representatives from groups must be selected based on their technical expertise, domain knowledge, credibility, and accessibility. There are several groups to be considered for conducting interviews:

(*i*) Entry level personnel: They may not have sufficient domain knowledge and experience, but may be very useful for fresh ideas and different views.

(*ii*) Mid-level stakeholders: They have better domain knowledge and experience of the project. They know the sensitive, complex and critical areas of the project. Hence, requirement engineers may be able to extract meaningful and useful information. Project leader should always be interviewed.

(*iii*) Managers or other Stakeholders: Higher level management officers like Vice-Presidents, General Managers, Managing Directors should also be interviewed. Their expectations may provide different but rich information for the software development.

(*iv*) Users of the software: This group is perhaps the most important because they will spend more time interacting with the software than any one else. Their information may be eye opener and may be original at times. Only caution required is that they may be biased towards existing systems.

Types of questions: Questions should be simple and short. Two or three questions rolled into one can lead to compound requirements statements that are difficult to interpret and test. It is important to prepare questions, but reading from the questionnaire or only sticking to it is not desirable. We should be open for any type of discussion and any direction of the interview. For the "result management system" we may ask:

- Are there any problems with the existing system ?
- Have you faced calculation errors in past ?
- What are the possible reasons of malfunctioning ?
- How many students are enrolled presently ?
- What are the possible benefits of computerising this system ?
- Are you satisfied with current processes and policies ?
- How are you maintaining the records of previous students ?
- What data, required by you, exists in other systems ?
- What problems do you want this system to solve ?

- Do you need additional functionality for improving the performance of the system ?
- What should be the most important goal of the proposed development ?

These questions will help to start the communication that is essential for understanding the requirements. At the end of this, we may have wide variety of expectations from the proposed software.

3.4.2 Brainstorming Sessions

Brainstorming is a group technique that may be used during requirements elicitation to understand the requirements. The group discussions may lead to new ideas quickly and help to promote creative thinking.

It is intended to generate lots of ideas, with full understanding that they may not be useful. The theory is that having a long list of requirements from which to choose is far superior to starting with a blank slate. Requirements in the long list can be categorized, prioritized, and pruned [ROBE02].

Brainstorming has become very popular and is being used by most of the companies. It promotes creative thinking, generates new ideas and provides plateform to share views, apprehensions expectations and difficulties of implementation. All participants are encouraged to say whatever ideas come to mind, whether they seem relevant or not. No one will be criticized for any idea, no matter how goofy it seems, as the responsibility of the participant is to generate views and not to vet them.

This group technique may be carried out with specialised groups like actual users, middle level managers etc., or with total stakeholders. Sometimes unnatural groups are created that may not be appriciated and are uncomfortable for participants. At times, only superficial responses may be gathered to technical questions. In order to handle such situations, a highly trained facilitator may be required. The facilitator may handle group bias and group conflicts carefully. The facilitator should also be cautious about individual egos, dominance and will be responsible for smooth conduct of brainstorming sessions. He or she will encourage the participants, ensure proper individual and group behaviour and help to capture the ideas. The facilitator will follow a published agenda and restart the creative process if it falters.

Every idea will be documented in such a way that everyone can see it. White boards, overhead transparencies or a computer projection system can be used to make it visible to every participant. After the session, a detailed report will be prepared and facilitator will review the report. Every idea will be written in simple english so that it conveys same meaning to every stakeholder. Incomplete ideas may be listed separately and should be discussed at length to make them complete ideas, if possible. Finally, a document will be prepared which will have list of requirements and their priority, if possible.

3.4.3 Facilitated Application Specification Technique

This approach is similar to brainstorming sessions and the objective is to bridge the expectation gap – a difference between what developers think they are supposed to build and what customers think they are going to get. In order to reduce expectation gap, a team oriented approach is developed for requirements gathering and is called Facilitated Application Specification Technique (FAST).

This approach encourages the creation of a joint team of customers and developers who work together to understand the expectations and propose a set of requirements. The basic guidelines for FAST are given below:

- Arrange a meeting at a neutral site for developers and customers.
- Establishment of rules for preparation and participation.
- Prepare an informal agenda that encourages free flow of ideas.
- Appoint a facilitator to control the meeting. A facilitator may be a developer, a customer, or an outside expert.
- Prepare a definition mechanism-board, flip charts, worksheets, wall stickies, etc.
- Participants should not criticize or debate.

FAST session preparations

Each FAST attendee is asked to make a list of objects that are:

(*i*) part of the environment that surrounds the system

(*ii*) produced by the system

(*iii*) used by the system.

In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (*e.g.*, cost, size) and performance criteria (*e.g.*, speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system [PRES2K].

Activities of FAST session

The activities during FAST session may have the following steps:

- Each participant presents his or her lists of objects, services, constraints, and performance for discussion. Lists may be displayed in the meeting by using board, large sheet of paper or any other mechanism, so that they are visible to all the participants.
- The combined lists for each topic are prepared by eliminating redundant enteries and adding new ideas.
- The combined lists are again discussed and consensus lists are finalised by the facilitator.
- Once the consensus lists have been completed, the team is divided into smaller subteams, each works to develop mini-specifications for one or more enteries of the lists.
- Each subteam then presents mini-specifications to all FAST attendees. After discussion, additions or deletions are made to the lists. We may get new objects, services, constraints, or performance requirements to be added to original lists.
- During all discussions, the team may raise an issue that cannot be resolved during the meeting. An issues list is prepared so that these ideas will be considered later.
- Each attendee prepares a list of validation criteria for the product/system and presents the list to the team. A consensus list of validation criteria is then created.

- A subteam may be asked to write the complete draft specifications using all inputs from the FAST meeting.

FAST is not a panacea of the problems encountered in early requirements elicitation but it helps to understand the requirements and bridge the expectation gap of develops and customers.

3.4.4 Quality Function Deployment

It is a quality management technique that helps to incorporate the voice of the customer. The voice is then translated into technical requirements. These technical requirements are documented and results is the software requirements and specification document. These requirements are further translated into design requirements. Here, customer satisfaction is of prime concern and thus QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the software engineering process [PRES2K]. Three types of requirements are identified [ZULT92]:

(*i*) **Normal requirements:** The objectives and goals of the proposed software are discussed with the customer. If this category of requirements (normal) are present, the customer is satisfied. Examples related to result management system might be: entry of marks, calculation of results, merit list report, failed students report, etc.

(*ii*) **Expected requirements:** These requirements are implicit to the software product and may be so obvious that customer does not explicitly state them. If such requirements are not present, customer will be dissatisfied with the software. Examples of expected requirements may be: protection from unauthorised access, some warning system for wrong entry of data, the feasibility for modification of any record only by a fool proof system for the identification of person alongwith date and time of modification, etc.

(*iii*) **Exciting requirements:** Some features go beyond the customer's expectations and prove to be very satisfying when present. Examples of exciting requirements for result management system may be: if an unauthorised access is noticed by the software, it should immediately shutdown all the processes and an E-mail is generated to the system administrator, an additional copy of important files is maintained and may be accessed by system administrator only, sophisticated virus protection system etc.

The QFD method has the following steps [ROBE02]:

(*i*) Identify all the stakeholders *e.g.*, customers, users, and developers. Also identify any initial constraints identified by the customer that affect requirements development.

(*ii*) List out requirements from customer ; inputs, considering different viewpoints. Requirements are expression of what the system will do, which is both perceptible and of value to customers. Some customer's expectations may be unrealistic or ambiguous and may be translated into realistic or unambiguous requirements if possible.

(*iii*) A value indicating a degree of importance, is assigned to each requirement. Thus, customer determines the importance of each requirement on a scale of 1 to 5 as given below:

5 points:	Very important
4 points:	important

3 points:	not important, but nice to have
2 points:	not important
1 point:	unrealistic, requires further exploration

Stakeholders will have their own unique set of criteria for determining the 'importance', or 'value' *of a requirement. It may be based on cost/benefit analysis particular to the project.*

Requirement engineers may review the final list of requirements and categorise like:

(*i*) it is possible to achieve

(*ii*) it should be deferred and the reason thereof

(*iii*) it is impossible and should be dropped from consideration.

The first category requirements will be implemented as per priority (Importance value) assigned with every requirement. If time and effort permits, second category requirements may be reviewed and few of them may be transferred to category first for implementation.

3.4.5 The Use case Approach

For many years, requirement engineers have used stories or scenarios to explain the interaction of a user with the proposed software system in order to gather the requirements. More recently, Ivar Jacobson and others [JACO99] formalised this into use case approach to requirements elicitation and modeling. Initially, use cases were designed for object oriented software development world, however, they can be applied to any project that follow any development approach because the user does not care how we develop the software. The focus on what the users need to do with the system is much more powerful than the traditional elicitation approach of asking users what they want the system to do [WIEG99].

This approach uses a combination of text and pictures in order to improve the understanding of requirements. The Use cases describe what of a system and not 'how'. They only give functional view of the system.

The terms use case, use case scenario, and use case diagram are often interchanged, but in fact they are different. Use cases are structured outline or templates for the description of user requirements, modeled in a structured language like english. Use case scenarios are unstructured descriptions of user requirements. Use case diagrams are graphical representations that may be decomposed into further levels of abstraction. The following components are used for the design of the use case approach.

Actor: An actor or external agent, lies outside the system model, but interacts with it in some way. An actor may be a person, machine, or an information system that is external to the system model. An actor is represented as stick figure and is not part of the system itself. Customers, users, external devices, or any external entity interacting with the system are treated as actors.

We should not confuse the actors with the devices they use. Devices are typically mechanisms that actors use to communicate with the system, but they are not actors themselves. We are writing this book on a computer, but the keyboad is not the user of the word processing program; but we are. Other devices, such as disk drives, tape drives, or communication equipment including printers have no place in use case diagram, although they are important

to the design of the system. The purpose of devices is to support some required behaviour of the system, but devices do not define the requirements of the system. Often systems must produce a printed report of information that it contains. We may want to show printer as an actor that then forwards the report to the real actor. This is not correct. Printer is not an actor; it is just a mechanism for conveying information [BITT03].

Cockburn [COCK01] distinguishes between primary and secondary actors. A primary actor is one having a goal requiring the assistance of the system. A secondary actor is one from which the system needs assistance.

Use cases: A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to deliver the services that satisfies the goal. It also includes possible variants of this sequence, *e.g.*, alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure to complete the service because of exceptional behaviour, error handling etc. The system is treated as a 'black box', and the interactions with the system, including responses, are as perceived from outside the system.

Thus, use cases capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals. A complete set of use cases specifies all the different ways to use the system, and therefore, defines all behaviour required of the system, bounding the scope of the system.

Use cases are written in an easy to understand structured narrative—the vocabulary of the domain. The users may validate the use cases and may involve in the process of gathering and defining the requirements [MALA01].

There is no standard use case template for writing use cases. The Jacobson et al. [JACO99] proposed a template for writing use cases and is given in Table 3.1(*a*). This template captures requirements in an effective way and is therefore becoming popular. Another similar template is also given in Table 3.1(*b*) which is also used by many organisations.

Use case guidelines

The following provides an outline of a process for creating use cases:

- Identify all the different users of the system.
- Create a user profile for each category of users, including all the roles the users play that are relevant to the system. For each role, identify all the significant goals the users have that the system will support. A statement of the system's value proposition is useful in identifying significant goals.
- Create a use case for each goal, following the use case template. Maintain the same level of abstraction throughout the use case. Steps in higher level use cases may be treated as goals for lower level (*i.e.*, more detailed), sub-use cases.
- Structure the use cases. Avoid over-structuring, as this can make the use cases harder to follow.
- Review and validate with users.

Table 3.1(a): Use case template

1.	**Brief Description.** Describe a quick background of the use case.
2.	**Actors.** List the actors that interact and participate in this use case.
3.	**Flow of Events.** 3.1. ***Basic flow.*** List the primary events that will occur when this use case is executed. 3.2. ***Alternative flows.*** Any subsidiary events that can occur in the use case should be separately listed. List each such event as an alternative flow. A use case can have as many alternative flows as required.
4.	**Special Requirements.** Business rules for the basic and alternative flows should be listed as special requirements in the use case narration. These business rules will also be used for writing test cases. Both success and failure scenarios should be described here.
5.	**Pre-conditions.** Pre-conditions that need to be satisfied for the use case to perform.
6.	**Post-conditions.** Define the different states in which you expect the system to be in, after the use case executes.
7.	**Extension Points.**

Table 3.1(b): Use case template

1.	**Introduction.** Describe brief purpose of the use case.
2.	**Actors.** List the actors that interact and participate in this use case.
3.	**Pre-condition.** Condition that need to be satisfied for the use case to execute.
4.	**Post-condition.** After the execution of the use case, different states of the systems are defined here.
5.	**Flow of Events.** 5.1. ***Basic flow.*** List the primary events that will occur when this use case is executed. 5.2. ***Alternate flow.*** Any other possible flow in this use case, if there, should be separately listed. A use case may have many alternate flows.
6.	**Special Requirements.** Business rules for the basic and alternate flows should be listed as special requirements. Both success and failure scenarios should be described.
7.	**Related use cases.** List the related use cases, if any.

Use case diagrams

A use case diagram visually represents what happens when an actor interacts with a system. Hence, a use case diagram captures the functional aspects of a system. The system is shown as a rectangle with the name of the system (or subsystem) inside, the actors are shown as stick figures (even the non human ones), the use cases are shown as solid bordered ovals labeled with the name of the use case, and relationships are lines or arrows between actors and use cases and/or between the use cases themselves. These components are given in Fig. 3.2.

Fig. 3.2: Components of use case diagram

Actors appear outside of the rectangle since they are external to the system. Use cases appear within the rectangle, providing functionality. A relationship or association is a solid line between an actor and each use case in which actor participates—the involvement can be any kind, not necessarily one of the actor initiating the use case functionality.

Fig. 3.2 shows an example of a use case diagram whose "Problem statement" is given in Table 3.2.

Table 3.2: Problem statement for railway reservation system

PROBLEM STATEMENT FOR RAILWAY RESERVATION SYSTEM

A software has to be developed for automating the manual railway reservation system. The system should be distributed in nature. It should be designed to provide functionalities as explained below:

1. **Reserve Seat:** A passenger should be able to reserve seats in the train. A reservation form is filled by the passenger and given to the clerk, who then checks for the availability of seats for the specified date of journey. If seats are available, then the entries are made in the system regarding the train name, train number, date of journey, boarding station, destination, person name, sex and total fare. Passenger is asked to pay the required fare and the tickets are printed. It the seats are not available then the passenger is informed.
2. **Cancel Reservation:** A passenger wishing to cancel a reservation is required to fill a form. The passenger then submits the form and the ticket to the clerk. The clerk then deletes the entries in the system and changes in the reservation status of that train. The clerk crosses the ticket by hand to mark as cancelled.
3. **Update Train Information:** Only the administrator enters any changes related to the train information like change in the train name, train number, train route etc. in the system.
4. **Report Generation:** Provision for generation of different reports should be given in the system. The system should be able to generate reservation chart, monthly train report etc.
5. **Login:** For security reasons all the users of the system are given a user *id* and a password. Only if the *id* and password are correct the user is allowed to enter the system.
6. **View Reservation Status:** All the users should be able to see the reservation status of the train online. The user needs to enter the train number and the pin number printed on his ticket so that the system can display his current reservation status like confirmed, RAC or Wait listed.
7. **View Train Schedule:** Provision should be given to see information related to the train schedules for the entire train network. The user should be able to see the train name, train number, boarding and destination stations, duration of journey etc.

Use cases should not be used to capture all the details of system. The granularity to which we define use cases in a diagram should be enough to keep the use case diagram uncluttered and readable, yet, be complete without missing significant aspects of the required

functionality. Design issues should not be discussed at all. Use cases are meant to capture "what" the system is, and not "how" the system will be designed or built. Hence use cases should be free of any design characteristics. If we end up defining design characteristics in a use case, we need to go back to the drawing board and start again.

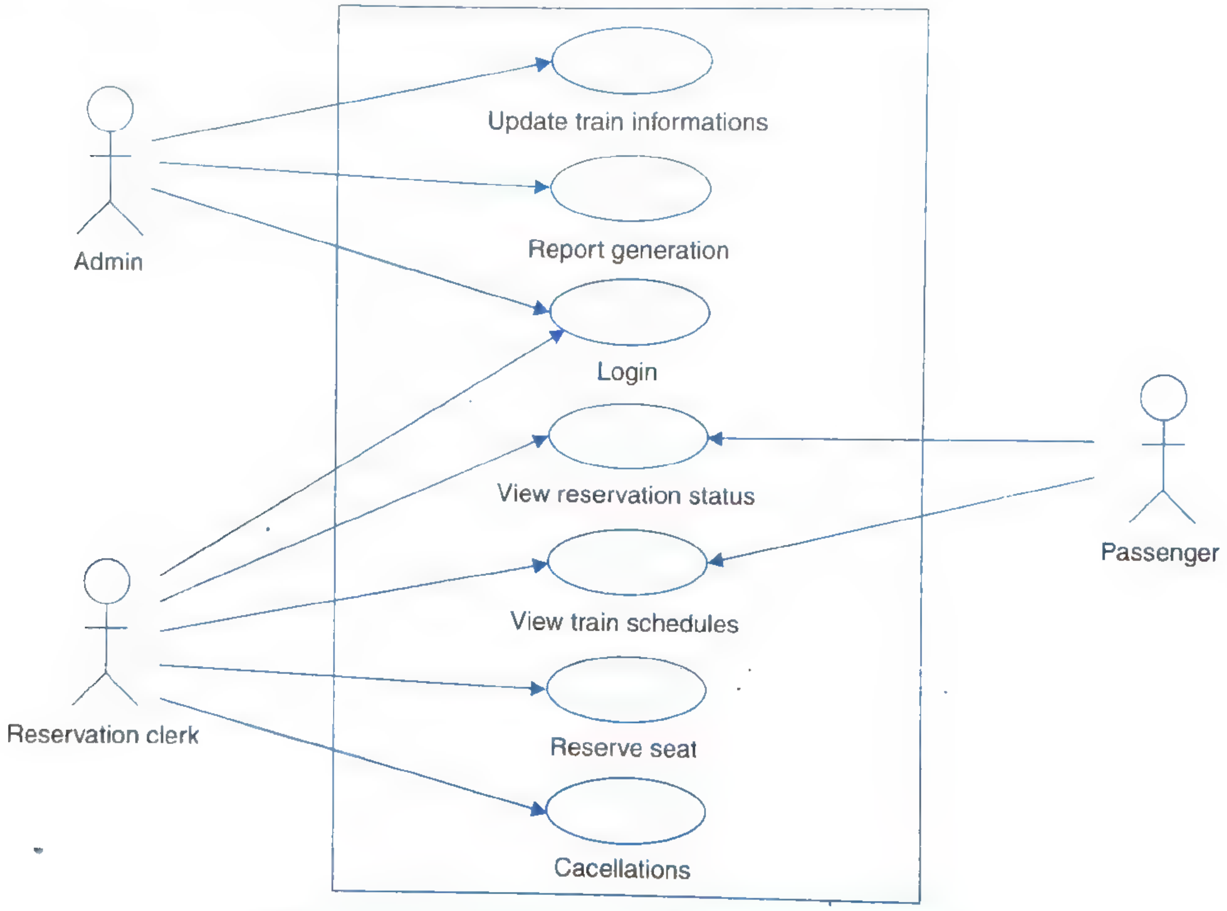


Fig. 3.3: Use case diagram for railway reservation system

3.5 REQUIREMENTS ANALYSIS

Requirements analysis is very important and essential activity after elicitation. We analyze, refine and scrutinize the gathered requirements in order to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the

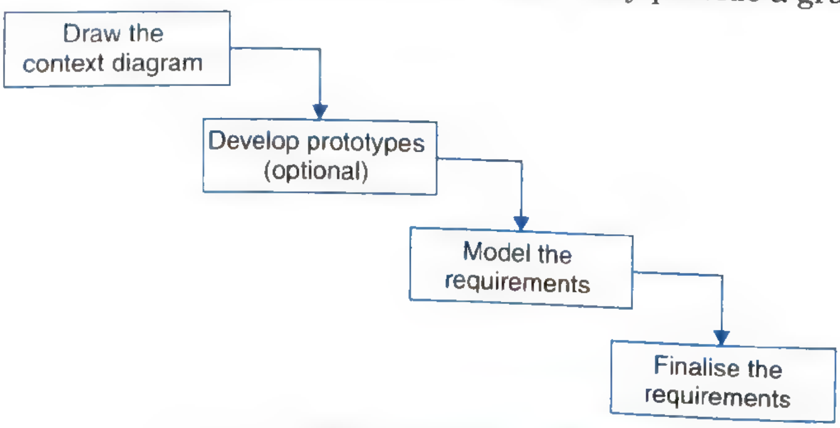
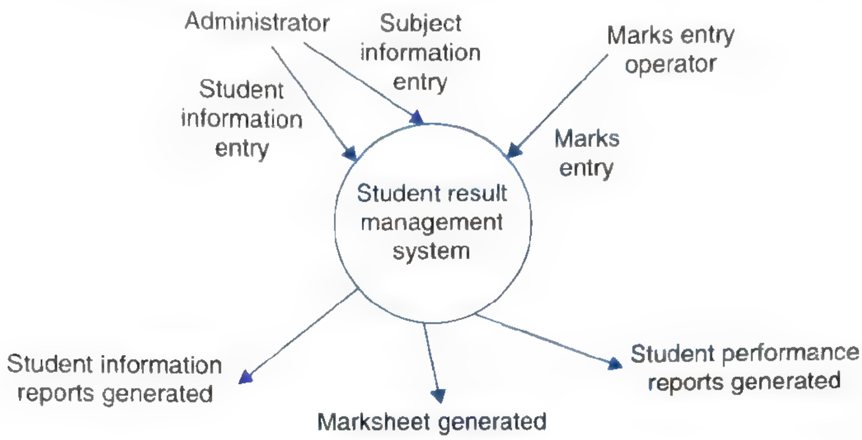


Fig. 3.4: Requirements analysis steps

entire system. After the completion of analysis, it is expected that the understandability of the project may improve significantly. Here, we may also interact with the customer to clarify points of confusion and to understand which requirements are more important than others. The various steps of requirements analysis are shown in Fig. 3.4.

(*i*) **Draw the context diagram:** The context diagram is a simple model that defines the boundaries and interfaces of the proposed system with the external world. It identifies the entities outside the proposed system that interact with the system. The context diagram of student result management system (as discussed earlier) is given below:



(*ii*) **Development of a prototype (optional):** One effective way to find out what the customer really wants is to construct a prototype, something that looks and preferably acts like a part of the system they say they want.

We can use their feedback to continuously modify the prototype until the customer is satisfied. Hence, prototype helps the client to visualise the proposed system and increase the understanding of requirements. When developers and users are not certain about some of the requirements, a prototype may help both the parties to take a final decision.

Some projects are developed for general market. In such cases, the prototype should be shown to some representative sample of the population of potential purchasers. Even though, persons who try out a prototype may not buy the final system, but their feedback may allow us to make the product more attractive to others. Some projects are developed for a specific customer under contract. On such projects, only that customer's opinion counts, so the prototype should be shown to the prospective users in the customer organisation.

The prototype should be built quickly and at a relatively low cost. Hence it will always have limitations and would not be acceptable in the final system. This is an optional activity. Although many organisations are developing prototypes for better understanding before the finalisation of SRS.

(*iii*) **Model the requirements:** This process usually consists of various graphical representations of the functions, data entities, external entities and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing and superfluous requirements. Such models include data flow diagrams, entity relationship diagrams, data dictionaries, state-transition diagrams etc.

(*iv*) **Finalise the requirements:** After modeling the requirements, we will have better understanding of the system behaviour. The inconsistencies and ambiguties have been identified and corrected. Flow of data amongest various modules has been analysed. Elicitation and analysis activities have provided better insight to the system. Now we finalise the analysed requirements and next step is to document these requirements in a prescribed format.

3.5.1 Data Flow Diagrams

Data flow diagrams (DFD) are used widely for modeling the requirements. They have been used for many years prior to the advent of computers. DFDs show the flow of data through a system. The system may be a company, an organisation, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or a bubble chart.

The following observations about DFDs are important [DAV190]:

1. All names should be unique. This makes it easier to refer to items in the DFD.
2. Remember that a DFD is not a flow chart. Arrows in a flow chart represent the order of events; arrows in DFD represent flowing data. A DFD does not imply any order of events.
3. Suppress logical decisions. If we ever have the urge to draw a diamond-shaped box in a DFD, suppress that urge! A diamond-shaped box is used in flow charts to represent decision points with multiple exit paths of which only one is taken. This implies an ordering of events, which makes no sense in a DFD.
4. Do not become bogged down with details. Defer error conditions and error handling until the end of the analysis.

Standard symbols for DFDs are derived from the electric circuit diagram analysis and are shown in Fig. 3.5 [SAGE90].

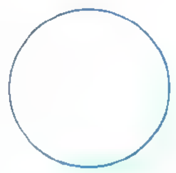
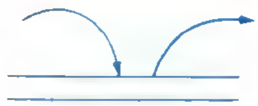
Symbol	*Name*	*Function*
	Data Flow	Used to connect processes to each other, to sources or sinks; the arrowhead indicates direction of data flow.
	Process	Performs some transformation of input data to yield output data.
	Source or Sink (External entity)	A source of system inputs or sink of system outputs.
	Data store	A repository of data; the arrowheads indicate net inputs and net outputs to store.

Fig. 3.5: Symbols for data flow diagrams

A circle (bubble) shows a process that transforms data inputs into data outputs. A curved line shows flow of data into or out of a process or data store. A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have element or group of elements. Source or sink is an external entity and acts as a source of system inputs or sink of system outputs.

Leveling

The DFD may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. A level-0 DFD, also called a fundamental system model or *context diagram* represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively [PRES2K]. Then the system is decomposed and represented as a DFD with multiple bubbles. Parts of the system represented by each of these bubbles are then decomposed and documented as more and more detailed DFDs. This process may be repeated at as many levels as necessary until the problem at hand is well understood. It is important to preserve the number of inputs and outputs between levels; this concept is called leveling by DeMacro. Thus, if bubble "A" has two inputs, x_1 and x_2, and one output y, then the expanded DFD, that represents "A" should have exactly two external inputs and one external output as shown in Fig. 3.6 [DEMA79, DAV190].

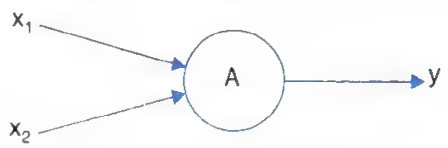


Fig. 3.6: Level-0 DFD

The level-0 DFD, also called context diagram of result management system is shown in Fig. 3.7. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flows may also need to be decomposed. Level-1 DFD of result management system is given in Fig. 3.8.

This provides a detailed view of requirements and flow of data from one bubble to the another.

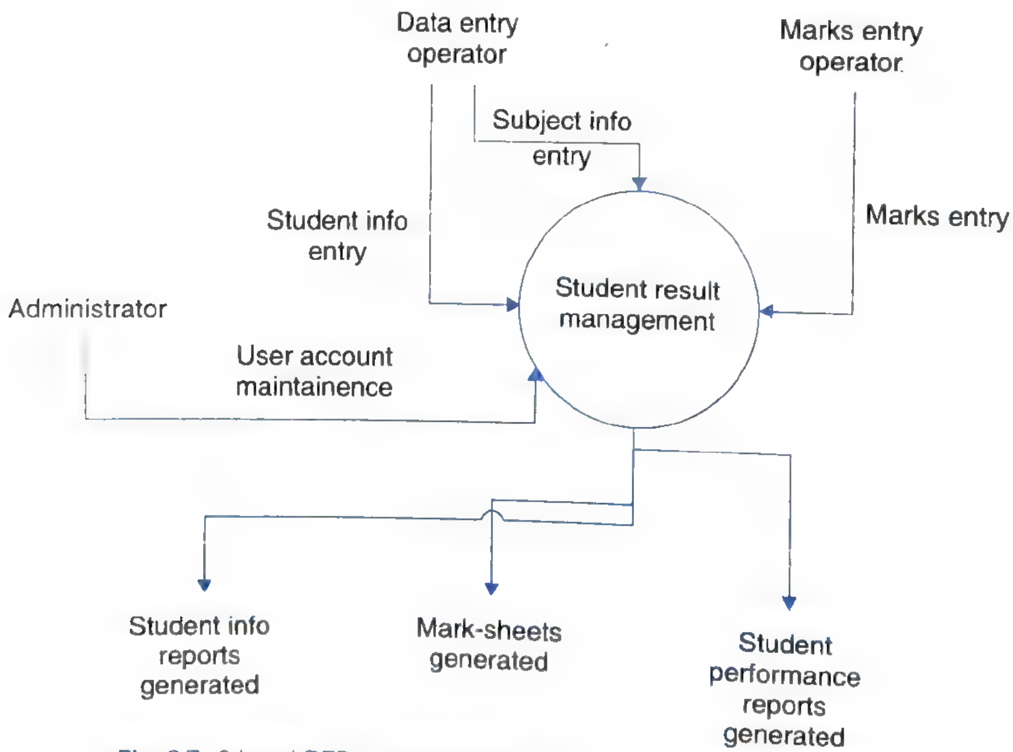


Fig. 3.7: 0-Level DFD or context diagram of result management system

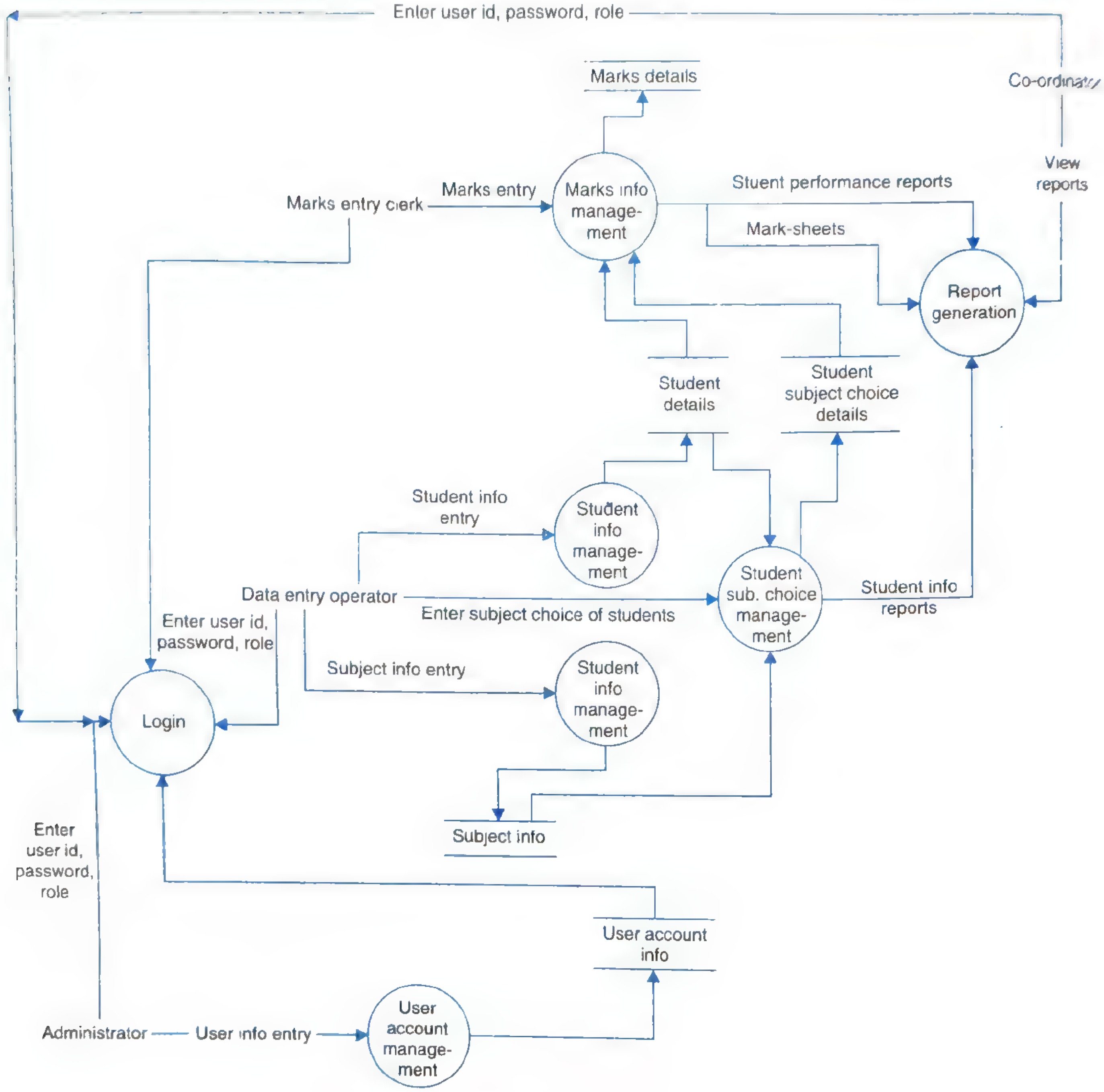


Fig. 3.8: Level-1 DFD of result management system

3.5.2 Data Dictionaries

Families of DFDs can become quite complex. One way to manage this complexity is to augment DFDs with data dictionaries (DD). Data dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developer use the same definitions and terminologies. Typical information stored includes:

- Name of the data item
- Aliases (other names for item)

- Description/purpose
- Related data items
- Range of values
- Data structure definition/form

The name of the data item is self-explanatory. Aliases include other names by which this data item is called *e.g.*, DEO for Data Entry Operator and DR for deputy Registrar. Description/Purpose is a textual description of what the data item is used for or why it exists. Related data items capture relationships between data items *e.g.*, total_marks must always equal to internal_marks plus external_marks.

Range of values records all possible values, *e.g.*, total marks must be positive and between 0 to 100. Data flows capture the names of the processes that generate or receive the data item. If data item is primitive, then data structure definition/form captures the physical structure of the data item. If the data is itself a data aggregate, then data structure definition/form captures the composition of the data items in terms of other data items [DAV190]. The mathematical operators used within the data dictionary are defined in Table 3.3 [DEMA79].

Table 3.3: Data dictionary notation and mathematical operators

Notation	*Meaning*
$x = a + b$	x consists of data elements a and b
$x = [a/b]$	x consists of either data element a or b
$x = a$	x consists of an optional data element a
$x = y\{a\}$	x consists of y or more occurrences of data element a
$x = \{a\}z$	x consists of z or fewer occurrences of data element a
$x = y\{a\}z$	x consists of some occurrences of data element a which are between y and z.

The data dictionary can be used to:

- Create an ordered listing of all data items.
- Create an ordered listing of a subset of data items.
- Find a data item name from a description.
- Design the software and test cases.

3.5.3 Entity-Relationship Diagrams

Another tool for requirement analysis is the entity-relationship diagram, often called as "E-R diagram" [CHEN76]. It is a detailed logical representation of the data for an organisation and uses three main constructs *i.e.*, data entities, relationships, and their associated attributes.

Entities

An entity is a fundamental thing of an organisation about which data may be maintained. An entity has its own identity, which distinguishes it from each other entity. An entity type is the description of all entities to which a common definition and common relationships and attributes apply.

Consider a university that offers both regular and distance education programmes. These programmes are offered to national and international students.

PROGRAMME and STUDENT are both entity types in this example. Regular and distance education are entities of PROGRAMME whereas national and international are entities of STUDENT.

We use capital letters in naming an entity type and in an ER diagram the name is placed inside a rectangle representing that entity as shown in Fig. 3.9.

Fig. 3.9: Two entity types in an E-R diagram

Relationships

A relationship is a reason for associating two entity types. These relationships are sometimes called binary relationships because they involve two entity types. Some forms of data model allow more than two entity types to be associated. A STUDENT is registered for a PROGRAMME. Relationships are represented by diamond notation in the E-R diagram as shown in Fig. 3.10.

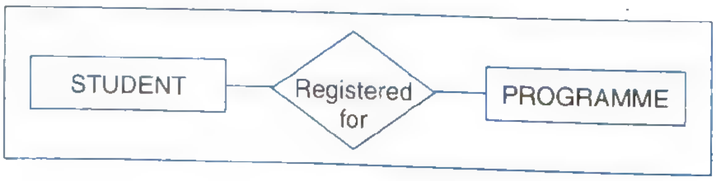


Fig. 3.10: Relationships added to ERD

We consider another example in which, a teaching department of a university in interested in tracking which subjects each of its students has completed. This leads to a relationship called "completes" between the STUDENT and SUBJECT entity types. This is shown in Fig. 3.11.

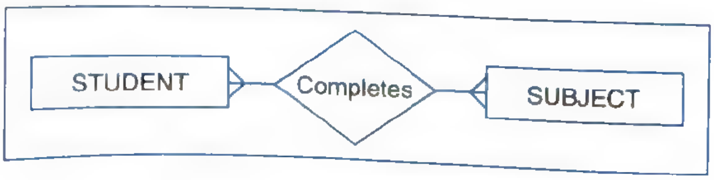


Fig. 3.11: Relationships in ERD

As indicated by the arrows, this is a many-to-many relationship. Each student may complete more than one subject, and more than one student may complete each subject.

Degree of relationships

The degree of a relationship is the number of entity types that participate in that relationship. Thus, relationship "completes" shown in Fig. 3.11 is of degree two, since there are two entity types: STUDENT and SUBJECT. The three most common relationships in E-R models are unary (degree 1), binary (degree 2), and ternary (degree three). Higher-degree relationships are possible, but they are rarely encountered in practice.

Unary relationship

This is also called recursive relationship. It is a relationship between the instances of one entity type. An instance is a single occurrence of an entity type. There may be many instances of an entity type. For example, there is one STUDENT entity type in universities, but there may be hundreds of instances of this entity type in the database. In Fig. 3.12, Is-Married-to is shown as one to one relationship between instances of the PERSON entity type. That is, each person may be currently married to one other person. In the second example, "Is friend of" is shown as one to many relationships between instances of the STUDENT entity type.

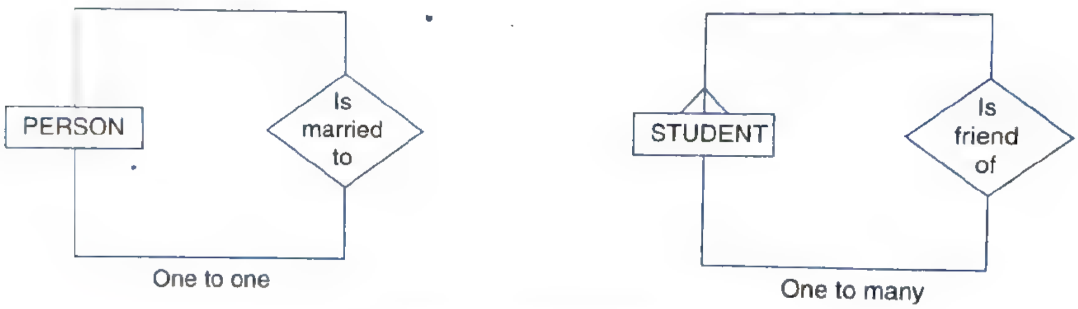


Fig. 3.12: Unary relationships

Binary relationship

It is a relationship between instances of two entity types and is the most common type of relationship encountered in E-R diagrams. Fig. 3.13 shows three examples.

The first (one to one) indicates that a STUDENT is assigned a STUDENT-ID, and each STUDENT-ID is assigned to a STUDENT. The second (one to many) indicates that a PROGRAMME may have many students, and each STUDENT belongs to only one PROGRAMME.

The third (many to many) shows that a STUDENT may register for more than one SUBJECT, and that each SUBJECT may have many STUDENT registrants.

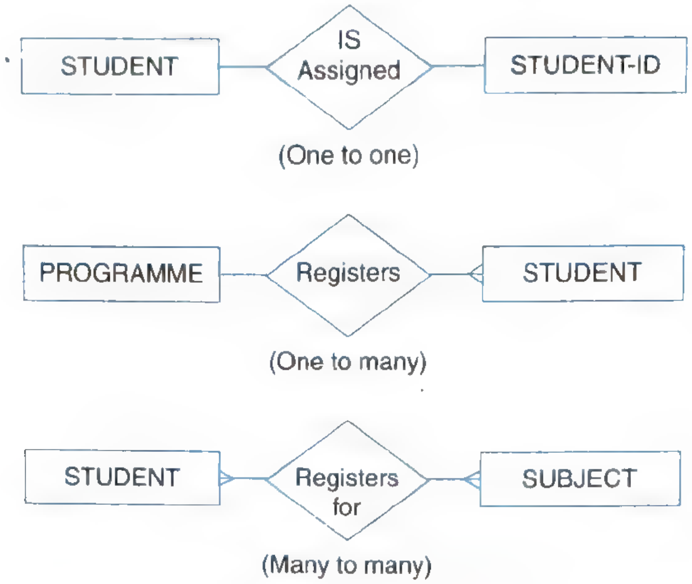


Fig. 3.13: Binary relationships

Ternary relationships

It is a simultaneous relationship amongst instances of three entity types. In Fig. 3.14, the relationship 'may have' provides the association of three entities *i.e.*, TEACHER, STUDENT and SUBJECT. All three entities are many-to many participants. There may be one or many participants in a ternary relationship.

In general, '*n*' entities can be related by the same relationship and is known as *n*-ary relationship.

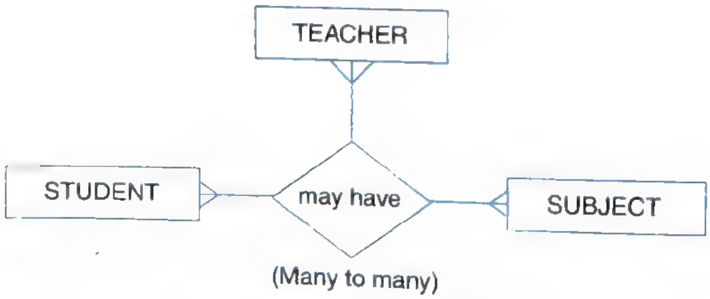


Fig. 3.14: Ternary relationship

Cardinalities and optionality

Suppose that there are two entity types, A and B, that are connected by a relationship. The cardinality of a relationship is the number of instances of entity B that can be associated with each instance of entity A.

Consider the example shown in Fig. 3.15. a student may register for many subjects.

Fig. 3.15: Use of cardinality

In the terminology, we have discussed so far, this example has "one-to many" relationship. Yet it may also be true that a subject may not have any student at specific instance of time. We need a more precise notation to indicate the range of cardinalities for a relationship.

The minimum cardinality of a relationship is the minimum number of instances of entity B that may be associated with each instance of entity A. If minimum number of students available for a subject is zero, we say that subject is an optional participant in the 'register for' relationship. When the minimum cardinality of a relationship is one, then we say entity B is a mandatory participant in the relationship. The maximum cardinality is the maximum number of instances. In our example, maximum is 'many'. The modified E-R diagram is given in Fig. 3.16. The zero through the line near the SUBJECT entity means a minimum cardinality of zero, while the crow's foot notation means a 'many' maximum cardinality.

Fig. 3.16: Modified ER diagram

Cardinality of relationships

It can be used to identify relationships between entity types. The cardinality of relationships is given in Fig. 3.17.

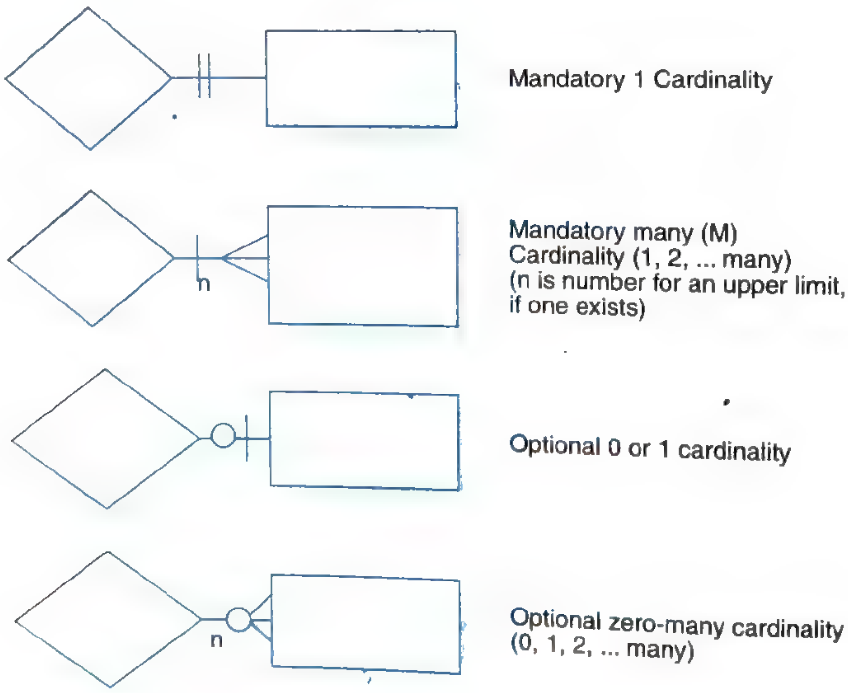


Fig. 3.17: Relationship cardinality [HOFF99]

Attributes

Each entity type has a set of attributes associated with it. An attribute is a property or characteristic of an entity that is of interest to the organization. Following are some typical entity types and associated attributes:

STUDENT: Student_ID, Student_Name, Address, Phone_Number

EMPLOYEE: Employee_ID, Employee_Name, Address.

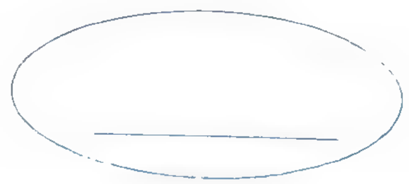
We use an initial capital letter, followed by lowercase letters, and nouns in naming an attribute. In E-R diagram, we can visually represent an attribute by placing its name as an ellipse with a line connecting it to the associated entity. Notation for attribute is

Attribute

Candidate keys and identifier

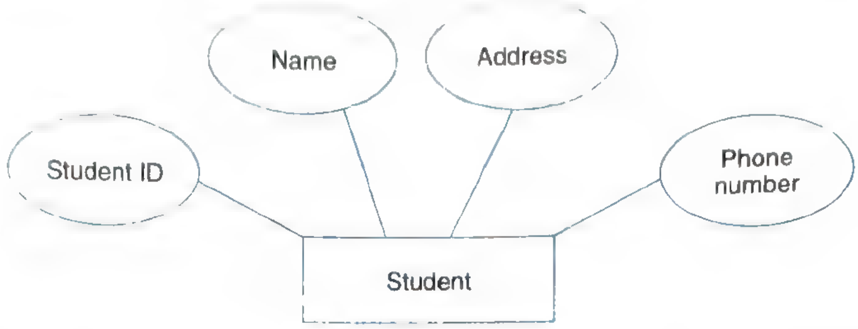
Every entity type must have an attribute or set of attributes that distinguishes one instance from other instances of the same type. A candidate key is an attribute (or combination of attributes) that uniquely identifies each instance of an entity type. A candidate key for a STUDENT entity type might be student_ID.

Some entities may have more than one candidate key. One candidate key for EMPLOYEE is Employee_ID, a second is the combination of Employee_Name and Address. If there is more than one candidate, the designer must choose one of the candidate keys as the identifier. An identifier is a candidate key that has been selected to be used as the unique characteristic for an entity type. Notation for identifier is



Identifier

The following diagram shows the representation for a STUDENT entity type using E-R notation [HOFF99].



Using entity relationship diagram to represent data is still an important technique today. It should not, however, be used in isolation, but together with techniques that fully represent the business objects we encounter every day.

The data flow diagram and the E-R diagram, each highlight a different aspect of the same system. As a consequence, there are one-to-one correspondences that must be checked to ensure that an E-R diagram and a data flow diagram are consistent over all applications. This suggests that it is desirable to use both methods such that we can view the logical issues from the two perspectives generated by these approaches.

3.5.4 Software Prototyping

Prototyping is the technique of constructing a partial implementation of a system so that customers, users, or developers can learn more about a problem or a solution to that problem. It is a partial implementation because if it were full implementation, it would be the system, not a prototype of it.

It allows users to explore and criticize proposed systems before undergoing the cost of a full-scale development. The field of prototyping software systems has emerged around two prototyping technologies, *i.e.*, throw-away and evolutionary. In throw-away approach, the prototype software is constructed in order to learn about the problem or its solution and is usually discarded after the desired knowledge is gained. In the evolutionary approach, the prototype is constructed in order to learn about the problem or its solution in successive steps. Once the prototype has been used and the requisite knowledge is gained, the prototype is then adapted to satisfy the, now better-understood, needs. The prototype is then used again, more is learned, and the prototype is re-adapted. This process repeats indefinitely until the prototype system satisfies all needs and thus evolves into the real system [DAV190]. Hence, in evolutionary prototyping the focus is on achieving functionality for demonstrating a portion of the system to the end user for feedback and system growth. The prototype emerges as the actual system downstream in the software life cycle. As with each iteration in development, functionality is added and then translated to an efficient implementation.

The benefits of developing a prototype early in the software process are [SOMM96]

1. Misunderstanding between software developers and customers may be identified as the system functions are demonstrated.
2. Missing user requirements may be detected.
3. Difficult-to-use or confusing user requirements may be identified and refined.

4. A working system is available quickly to demonstrate the feasibility and usefulness of the application to management.
5. The prototype serves as a basis for writing the specification of the system.

Software prototyping taxonomy

A range of possibilities exists for prototyping software systems. Any form of prototyping is perceived better than not prototyping at all. Several taxonomies have been proposed and served as basis for prototyping [RATC88, HOOP89, CERI86, CARE90, HEKM87]. However, the two most popular prototyping approaches mentioned earlier are briefly described as:

1. Throw-away prototyping: In this approach, prototype is constructed with the idea that it will be discarded, after the analysis is complete, and the final system is built from the scratch. This prototype is generally built quickly so as to enable the user to rapidly interact with the requirements determination early and thoroughly. Since the prototype will ultimately be discarded, it need not necessarily be fast operating, maintainable and having extensive fault tolerant capabilities.

During the requirement phase, a quick and dirty throw-away prototype can be constructed and given to user in order to determine the feasibility of a requirement, validate that a particular function is really necessary, uncover missing requirements and determine the viability of a user interface. During preliminary and detailed design, a quick and dirty prototype can be built to give a feeling and overview of final system to the user. Here, development of prototype should be quick, because its advantage exists only if results from its use are available in a timely fashion. It can be dirty because there is no justification for building quality into a product that will be discarded. Among the dirty characteristics to be considered are no design, no comments, no test plans, no idea about coupling and cohesion etc.

The most common steps for this approach are: (*i*) Writing a preliminary SRS (*ii*) implementing the prototype based on those requirements (*iii*) achieving user experience with the prototype (*iv*) Writing the real SRS and then (*v*) developing the real product.

2. Evolutionary prototyping: In this approach, the prototype is built with the idea that it will eventually be converted into the final system. It will not be built in a "dirty" fashion. The evolutionary prototype evolves into the final product, and thus it must exhibit all the quality attributes of the final product and must follow the traditional life cycle. It is required to deploy the product, obtain experience using it, then based on that experience go back and redo the requirements, redesign, recode, retest, and redeploy. After gaining more experience, it is time to repeat the entire process again. This ensures the creation of all necessary documents and the presence of all necessary reviews. In fact, the only shortcuts that should be taken in building evolutionary prototypes are (*i*) building only those parts of the product that are understood (leaving other parts to later generations of the prototype) (*ii*) lowering the importance of performance. Using this will increase the probability that version $i + 1$ will meet user's real needs because users have already used version i and supplied feedback on its performance.

The differences between these two approaches are given below:

Sr. No.	*Approach and characteristics*	*Throwaway*	*Evolutionary*
1	Development approach	Quick and dirty , No rigor	No Sloppiness, Rigorous
2	What to build	Build only difficult parts	Build understood parts first and build on solid foundations
3	Design drivers	Optimize development time	Optimize modifiability
4	Ultimate Goal	Throw it away	Evolve it

Prototyping pitfalls

Prototyping has not been as successful as anticipated in some organisations for a variety of reasons [TOZE87]. Training, efficiency, applicability, and behaviour can each have a negative impact on using software prototyping techniques.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort. Prototyping can have execution inefficiencies with the associated tools and this question may be argued as a negative aspect of prototyping.

This new approach of providing feedback early to the end user may result in a problem related to the behavior of the end user and developers. An end user with a previously unfortunate system development effort can be biased in future interactions with development teams.

Prototyping opportunities

Not to prototype at all should simply not be an option in software development. The benefits of software prototyping are obvious and established. The end user cannot throw the ambiguous and incomplete software needs and expect the development team to return the finished software system after some period of time with no problems in the deliverables.

One of the major problems incorporating this technology is the large investment that exists in software system maintenance. The idea of completely re-engineering an existing software system with current technology is not feasible. There is, however, a threshold that exists where the expected life span of a software system justifies that the system would be better maintained after being re-engineered in this technology. Total re-engineering should be planned rather than as a reaction to a crisis situation. At minimum, prototyping technology could be used on critical portion of an existing software system. This minimal approach could be used as a means to transition an organisation to total re-engineering.

Software prototyping must be integrated within an organisation through training, case studies, and library development. In situations where this full range of commitment to this technology is lacking, e.g. only developers training is provided, when problems begin to arise in using the technology a normal reaction of management is to revert back to what has worked in the past.

The end user involvement becomes enhanced when changes in requirements can be prototyped and agreed to before any development proceeds. Similarly, during development of

the actual system or even later into maintenance, should the requirements change; the prototype is enhanced and agreed to before the actual changes become confirmed.

3.6 REQUIREMENTS DOCUMENTATION

Requirements documentation is very important activity after the requirements elicitation and analysis. This is the way to represent requirements in a consistent format. Requirements document is called Software Requirements Specification (SRS).

The SRS is a specification for a particular software product, program or set of programs that performs certain functions in a specific environment. It serves a number of purposes depending on who is writing it. First, the SRS could be written by the customer of a system. Second, the SRS could be written by a developer of the system. The two scenarios create entirely different situations and establish entirely different purposes for the document. First case, SRS is used to define the needs and expectations of the users. The second case, SRS is written for different purpose and serve as a contract document between customer and developer.

This reduces the probability of the customer being disappointed with the final product. The SRS written by developer (second case) is of our interest and discussed in the subsequent sections.

3.6.1 Nature of the SRS

The basic issues that SRS writer (s) shall address are the following:

1. **Functionality:** What the software is supposed to do?
2. **External interfaces:** How does the software interact with people, the system's hardware, other hardware, and other software?
3. **Performance:** What is the speed, availability, response time, recovery time, etc. of various software functions?
4. **Attributes:** What are the considerations for portability, correctness, maintainability, security, reliability etc.?
5. **Design constraints imposed on an implementation:** Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment (s) etc.?

Since the SRS has a specific role to play in the software development process, SRS writer(s) should be careful not to go beyond the bounds of that role. This means the SRS

1. should correctly define all the software requirements. A software requirement may exist because of the nature of the task to be solved or because of a special characteristic of the project.
2. should not describe any design or implementation details. These should be described in the design stage of the project.
3. should not impose additional constraints on the software. These are properly specified in other documents such as a software quality assurance plan.

Therefore, a properly written SRS limits the range of valid designs, but does not specify any particular design.

3.6.2 Characteristics of a good SRS

The SRS should be:

— Correct

— Unambiguous

— Complete

— Consistent

— Ranked for importance and/or stability

— Verifiable

— Modifiable

— Traceable

Each of the above mentioned characteristics is discussed below: [THAY97, IEEE87, IEEE97].

Correct

The SRS is correct if, and only if; every requirement stated therein is one that the software shall meet. There is no tool or procedure that assures correctness. If the software must respond to all button presses within 5 seconds and the SRS stated that "the software shall respond to all buttons presses within 10 second", then that requirement is incorrect.

Unambiguous

The SRS is unambiguous if, and only if; every requirement stated therein has only one interpretation. Each sentence in the SRS should have the unique interpretation. Imagine that a sentence is extracted from the SRS, given to ten people who are asked for their interpretation. If there is more than one such interpretation, then that sentence is probably ambiguous.

In cases, where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific. The SRS should be unambiguous to both those who create it and to those who use it. However, these groups often do not have the same background and therefore do *not tend to describe* software requirements in the same way.

Requirements are often written in natural language (for example, english). Natural language is inherently ambiguous. A natural language SRS should be reviewed by an independent party to identify ambiguous use of a language so that it can be corrected. This can be avoided by using a particular requirement specification language. Its language processors automatically detect many lexical, syntactic, and semantic errors. Disadvantage is the time required to learn the language which may also not be understandable to the customers/users. Moreover, these languages tend to be better at expressing only certain types of requirements and addressing certain types of systems.

Complete

The SRS is complete if, and only if; it includes the following elements:

1. All significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces.

2. Definition of their responses of the software to all realizable classes *of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid values.*
3. Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

Consistent

The SRS is consistent if, and only if, no subset of individual requirements described in it conflict. There are three types of likely conflicts in the SRS:

1. The specified characteristics of real-world objects may conflict. For example.
   (*a*) The format of an output report may be described in one requirement as tabular but in another as textual.
   (*b*) One requirement may state that all lights shall be green while another states that all lights shall be blue.
2. There may be logical or temporal conflict between two specified actions, for example,
   (*a*) One requirement may specify that the program will add two inputs and another may specify that the program will multiply them.
   (*c*) One requirement may state that "A" must always follow "B", while another requires that "A and B" occur simultaneously.
3. Two or more requirements may describe the same real-world object but use different terms for that object. For example, a program's request for a user input may be called a "prompt" in one requirement and a "cue" in another. The use of standard terminology and definitions promotes consistency.

Ranked for importance and/or stability

The SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

Typically, all requirements are not equally important. Some requirements may be essential, especially for life critical applications, while others may be desirable. Each requirement should be identified to make these differences clear and explicit. Another way to rank requirements is to distinguish classes of requirements as essential, conditional and optional.

Verifiable

The SRS is verifiable, if and only if, every requirement stated therein is verifiable. A requirement is verifiable, if and only if, there exists some finite cost-effective process with which a person or machine can check that the software meets the requirements. In general any ambiguous requirement is not verifiable.

Non-verifiable requirements include statement, such as "works well", "good human interface", and "shall usually happen". These requirements cannot be verified because it is impossible to define the terms "good", "well", or "usually". The statement that "the program

shall never enter an infinite loop" is non-verifiable because the testing of this quality is theoretically impossible.

An example of a verifiable statement is "output of the program shall be produced within 20 seconds of event × 60% of the time; and shall be produced within 30 seconds of event × 100% of the time." This statement can be verified because it uses concrete terms and measurable quantities.

If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

Modifiable

The SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.

The requirements should not be redundant. Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. For instance, a requirement may be altered in only one of the places out of the many places where it appears.

The SRS then becomes inconsistent. Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

Traceable

The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended.

1. Backward traceability: This depends upon each requirement explicitly referencing its source in earlier documents.
2. Forward traceability: This depends upon each requirement in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

3.6.3 Organization of the SRS

The Institute of Electrical and Electronics Engineers (IEEE) has published guidelines and standards to organize an SRS document [IEEE87, IEEE94]. Different projects may require their requirements to be organized differently, that is, there is no one method that is suitable for all projects. It provides different ways of structuring the SRS. The first two sections of the SRS are the same in all of them. The specific tailoring occurs in section 3 entitled "specific requirements". The general organisation of an SRS is given in Fig. 3.18 [IEEE93].

1. Introduction
    1.1 Purpose
    1.2 Scope
    1.3 Definitions, Acronyms, and Abbreviations
    1.4 References
    1.5 Overview
2. The Overall Description
    2.1 Product Perspective
        2.1.1 System Interfaces
        2.1.2 Interfaces
        2.1.3 Hardware Interfaces
        2.1.4 Software Interfaces
        2.1.5 Communications Interfaces
        2.1.6 Memory Constraints
        2.1.7 Operations
        2.1.8 Site Adaptation Requirements
    2.2 Product Functions
    2.3 User Characteristics
    2.4 Constraints
    2.5 Assumptions and Dependencies.
    2.6 Apportioning of Requirements
3. Specific Requirements
    3.1 External interfaces
    3.2 Functions
    3.3 Performance Requirements
    3.4 Logical Database Requirements
    3.5 Design Constraints
        3.5.1 Standards Compliance
    3.6 Software System Attributes
        3.6.1 Reliability
        3.6.2 Availability
        3.6.3 Security
        3.6.4 Maintainability
        3.6.5 Portability
    3.7 Organising the Specific Requirements
        3.7.1 System Mode
        3.7.2 User Class
        3.7.3 Objects
        3.7.4 Feature
        3.7.5 Stimulus
        3.7.6 Response
        3.7.7 Functional Hierarchy
    3.8 Additional Comments
4. Change Management Process
5. Document Approvals
6. Supporting Information

Fig. 3.18: Organisation of SRS [IEEE-std. 830-1993]

1. Introduction

The following subsections of the Software Requirements Specifications (SRS) document should provide an overview of the entire SRS.

1.1 Purpose

Identify the purpose of this SRS and its intended audience. In this subsection, describe the purpose of the particular SRS and specify the intended audience for the SRS.

1.2 Scope

In this subsection:

(*i*) Identify the software product(s) to be produced by name

(*ii*) Explain what the software product(s) will, and, if necessary, will not do

(*iii*) Describe the application of the software being specified, including relevant benefits, objectives, and goals

(*iv*) Be consistent with similar statements in higher-level specifications if they exist.

1.3 Definitions, Acronyms, and Abbreviations

Provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendices in the SRS or by reference to documents. This information may be provided by reference to an Appendix.

1.4 References

In this subsection:

(*i*) Provide a complete list of all documents referenced elsewhere in the SRS

(*ii*) Identify each document by title, report number (if applicable), date, and publishing organisation

(*iii*) Specify the sources from which the references can be obtained.

This information can be provided by reference to an appendix or to another document.

1.5 Overview

In this subsection:

(*i*) Describe what the rest of the SRS contains

(*ii*) Explain how the SRS is organised.

2. The Overall Description

Describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in section 3, and makes them easier to understand.

2.1 Product Perspective

Put the product into perspective with other related products. If the product is independent and totally self-contained, it should be so stated here. If the SRS defines a product that is a component of a larger system, as frequently occurs, then this subsection relates the require-

ments of the larger system to functionality of the software and identifies interfaces between that system and the software.

A block diagram showing the major components of the large system, interconnections, and external interfaces can be helpful.

The following subsections describe how the software operates inside various constraints.

2.1.1 System Interfaces

List each system interface and identify the functionality of the software to accomplish the system requirement and the interface description to match the system.

2.1.2 Interfaces

Specify:

(*i*) The logical characteristics of each interface between the software product and its users.

(*ii*) All the aspects of optimizing the interface with the person who must use the system.

2.1.3 Hardware Interfaces

Specify the logical characteristics of each interface between the software product and the hardware components of the system. This includes configuration characteristics. It also covers such matters as what devices are to be supported, how they are to be supported and protocols.

2.1.4 Software Interfaces

Specify the use of other required software products and interfaces with other application systems. For each required software product, include:

(*i*) Name

(*ii*) Mnemonic

(*iii*) Specification number

(*iv*) Version number

(*v*) Source

For each interface, provide:

(*i*) Discussion of the purpose of the interfacing software as related to this software product

(*ii*) Definition of the interface in terms of message content and format.

2.1.5 Communications Interfaces

Specify the various interfaces to communications such as local network protocols, etc.

2.1.6 Memory Constraints

Specify any applicable characteristics and limits on primary and secondary memory.

2.1.7 Operations

Specify the normal and special operations required by the user such as:

(*i*) The various modes of operations in the user organisation

(*ii*) Periods of interactive operations and periods of unattended operations

(*iii*) Data processing support functions

(*iv*) Backup and recovery operations.

2.1.8 Site Adaptation Requirements

In this section:

(*i*) Define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode

(*ii*) Specify the site or mission-related features that should be modified to adapt the software to a particular installation.

2.2 Product Functions

Provide a summary of the major functions that the software will perform. Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product.

For clarity:

(*i*) The functions should be organised in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.

(*ii*) Textual or graphic methods can be used to show the different functions and their relationships. Such a diagram is not intended to show a design of a product but simply shows the logical relationships among variables.

2.3 User Characteristics

Describe those general characteristics of the intended users of the product including educational level, experience, and technical expertise. Do not state specific requirements but rather provide the reasons why certain specific requirements are later specified in sections 3.

2.4 Constraints

Provide a general description of any other items that will limit the developer's options. These can include:

(*i*) Regulatory policies

(*ii*) Hardware limitations (for example, signal timing requirements)

(*iii*) Interface to other applications

(*iv*) Parallel operation

(*v*) Audit functions

(*vi*) Control functions

(*vii*) Higher-order language requirements

(*viii*) Signal handshake protocols (for example, XON-XOFF, ACK-NACK)

(*ix*) Reliability requirements

(*x*) Criticality of the application

(*xi*) Safety and security considerations.

2.5 Assumptions and Dependencies

List each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operating system would be available on the hardware designated for the software product. If, in fact, the operating system were not available, the SRS would then have to change accordingly.

2.6 Apportioning of Requirements

Identify requirements that may be delayed until future versions of the system.

3. Specific Requirements

This section contains all the software requirements at a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements. Throughout this section, every stated requirement should be externally perceivable by users, operators, or both external systems. These requirements should include at a minimum a description of every input into the system, every output from the system and all functions performed by the system in response to an input or in support of an output. The following principles apply:

(*i*) Specific requirements should be stated with all the characteristics of a good SRS

- correct
- unambiguous
- complete
- consistent
- ranked for importance and/or stability
- verifiable
- modifiable
- traceable

(*ii*) Specific requirements should be cross-referenced to earlier documents that relate

(*iii*) All requirements should be uniquely identifiable

(*iv*) Careful attention should be given to organizing the requirements to maximize readability.

Before examining specific ways of organising the requirements it is helpful to understand the various items that comprise requirements as described in the following sebsections.

3.1 External Interfaces

This contains a detailed description of all inputs into and outputs from the software system. It complements the interface descriptions in *section 2 but does* not repeat information there.

It contains both content and format as follows:

- Name of item
- Description of purpose
- Source of input or destination of output
- Valid range, accuracy and/or tolerance
- Units of measure

- Timing
- Relationships to other inputs/outputs
- Screen formats/organisation
- Window formats/organisation
- Data formats
- Command formats
- End messages.

3.2 Functions

Functional requirements define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as "shall" statements starting with "The system shall...

These include:

- Validity checks on the inputs
- Exact sequence of operations
- Responses to abnormal situation, including
  - Overflow
  - Communication facilities
  - Error handling and recovery
- Effect of parameters
- Relationship of outputs to inputs, including
  - Input/Output sequences
  - Formulas for input to output conversion.

It may be appropriate to partition the functional requirements into sub-functions or sub-processes. This does not imply that the software design will also be partitioned that way.

3.3 Performance Requirements

This subsection specifies both the static and the dynamic numerical requirements placed on the software or on human interaction with the software, as a whole. Static numerical requirements may include:

(*i*) The number of terminals to be supported

(*ii*) The number of simultaneous users to be supported

(*iii*) Amount and type of information to be handled

Static numerical requirements are sometimes identified under a separate section entitled capacity.

Dynamic numerical requirements may include, for example, the number of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms:

For example,

95% of the transactions shall be processed in less than 1 second rather than,

An operator shall not have to wait for the transaction to complete.

(**Note:** Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function).

3.4 Logical Database Requirements

This section specifies the logical requirements for any information that is to be placed into a database. This may include:

- Types of information used by various functions
- Frequency of use
- Accessing capabilities
- Data entities and their relationships
- Integrity constraints
- Data retention requirements.

3.5 Design Constraints

Specify design constraints that can be imposed by other standards, hardware limitations, etc.

3.5.1 Standards Compliance

Specify the requirements derived from existing standards or regulations. They might include:

(*i*) Report format
(*ii*) Data naming
(*iii*) Accounting procedures
(*iv*) Audit Tracing

For example, this could specify the requirement for software to processing activity. Such traces are needed for some applications to meet minimum regulatory or financial standard. An audit trace requirement may, for example, state that all changes to a payroll database must be recorded in a trace file with before and after values.

3.6 Software System Attributes

There are a number of quality attributes of software that can serve as requirements. It is important that required attributes be specified so that their achievement can be objectively verified. Fig. 3.19 has the definitions of the quality attributes of the software discussed in this subsection [ROBE02]. The following items provide a partial list of examples.

3.6.1 Reliability

Specify the factors required to establish the required reliability of the software system at time of delivery.

3.6.2 Availability

Specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery, and restart.

3.6.3 Security

Specify the factors that would protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to:

- Utilize certain cryptographic techniques
- Keep specific log or history data sets
- Assign certain functions to different modules
- Restrict communications between some areas of the program
- Check data integrity for critical variables.

3.6.4 Maintainability

Specify attributes of software that relate to the ease of maintenance of the software itself. There may be some requirement for certain modularity, interfaces, complexity, etc. Requirements should not be placed here just because they are thought to be good design practices.

3.6.5 Portability

Specify attributes of software that relate to the ease of parting the software to other host machines and/or operating systems. This may include:

- Percentage of components with host-dependent code
- Percentage of code that is host dependent
- Use of a proven portable language
- Use of a particular compiler or language subset
- Use of a particular operating system.

S. No.	*Quality Attributes*	*Definition*
1.	Correctness	extent to which program satisfies specifications, fulfills user's mission objectives
2.	Efficiency	amount of computing resources and code required to perform function
3.	Flexibility	effort needed to modify operational program
4.	Interoperability	effort needed to couple one system with another
5.	Reliability	extent to which program performs with required precision
6.	Reusability	extent to which it can be reused in another application
7.	Testability	effort needed to test to ensure performance as intended
8.	Usability	effort required to learn, operate, prepare input, and interpret output
9.	Maintainability	effort required to locate and fix an error during operation
10.	Portability	effort needed to transfer from one hardware or software environment to another.
11.	Integrity/security	extent to which access to software or data by unauthorised people can be controlled.

Fig. 3.19: Definitions of quality attributes

3.7 Organising the Specific Requirements

For anything but trivial systems the detailed requirements tend to be extensive. For this reason, it is recommended that careful consideration be given to organising these in a manner optimal for understanding. There is no one optimal organisation for all systems. Different classes of systems lend themselves to different organisations of requirements. Some of these organisations are described in the following subclasses.

3.7.1 System mode

Some systems behave quite differently depending on the mode of operation. When organising by mode there are two possible outlines. The choice depends on whether interfaces and performance are dependent on mode.

3.7.2 User class

Some systems provide different sets of functions to different classes of users.

3.7.3 Objects

Objects are real-world entities that have a counterpart within the system. Associated with each object is a set of attributes and functions. These functions are also called services, methods, or processes. Note that sets of objects may share attributes and services. These are grouped together as classes.

3.7.4 Feature

A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. Each feature is generally described as sequence of stimulus-response pairs.

3.7.5 Stimulus

Some systems can be best organised by describing their functions in terms of stimuli.

3.7.6 Response

Some systems can be best organised by describing their functions in support of the generation of a response.

3.7.7 Functional hierarchy

When none of the above organisational schemes prove helpful, the overall functionality can be organised into a hierarchy of functions organised by either common inputs, common outputs, or common internal data access. Data flow diagrams and data dictionaries can be used to show the relationships between and among the functions and data.

3.8 Additional Comments

Whenever a new SRS is contemplated, more than one of the organisational techniques given in 3.7 may be appropriate. In such cases, organise the specific requirements for multiple hierarchies tailored to the specific needs of the system under specification.

There are many notations, methods, and automated support tools available to aid in the documentation of requirements. For the most part, their usefulness is a function of organisation. For example, when organising by mode, finite state machines or state charts may prove helpful, when organising by object, object-oriented analysis may prove helpful; when organising by feature, stimulus-response sequences may prove helpful; when organising by functional hierarchy, data flow diagrams and data dictionaries may prove helpful.

In any of the outlines below, those sections called "Functional Requirement i" may be described in native language, in pseudocode, in a system definition language, or in four sub-sections titled: Introduction, Inputs, Processing, Outputs.

4. Change Management Process

Identify the change management process to be used to identify, log, evaluate, and update the SRS to reflect changes in project scope and requirements.

5. Document Approval

Identify the approvers of the SRS document. Approver's name, signature, and date should be used.

6. Supporting Information

The supporting information makes the SRS easier to use. It includes:

- Table of Contents
- Index
- Appendices

The appendices are not always considered part of the actual requirements specification and are not always necessary. They may include:

(*a*) Sample I/O formats, descriptions of cost analysis studies, results of user surveys

(*b*) Supporting or background information that can help the readers of the SRS

(*c*) A description of the problems to be solved by the software

(*d*) Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements.

When appendices are included, the SRS should explicitly state whether or not the appendices are to be considered part of the requirements.

Tables on the following pages provide alternate ways to structure section 3 on the specific requirements.

Outline for SRS Section 3
Organised by Mode: Version 1

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 Mode 1
    - 3.2.1.1 Functional requirements 1.1 ......
    - 3.2.1.n Functional requirements 1.n
  - 3.2.2 Mode 2 ......
  - 3.2.m Mode m
    - 3.2.m.1 Functional requirement m.1 ......
    - 3.2.m.n Functional requirement m.n
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organised by Mode: Version 2

3 Specific Requirements
- 3.1 Functional Requirements
  - 3.1.1 Mode 1
    - 3.1.1.1 External interfaces
    - 3.1.1.1 User Interfaces
    - 3.1.1.2 Hardware interfaces
    - 3.1.1.3 Software interfaces
    - 3.1.1.4 Communications interfaces
    - 3.1.1.2 Functional Requirement
      - 3.1.1.2.1 Functional requirement 1 ......
      - 3.1.1.2.1.n Functional requirement n
    - 3.1.1.3 Performance
  - 3.1.2 Mode 2 ......
  - 3.1.m Mode m
- 3.2 Design constraints
- 3.3 Software system attributes
- 3.4 Other requirements.

Outline for SRS Section 3
Organised by User Class

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 User class 1
    - 3.2.1.1 Functional requirements 1.1 ......
    - 3.2.1.*n* Functional requirement 1.*n*
  - 3.2.2 User class 2 ......
  - 3.2.*m* User class *m*
    - 3.2.*m*.1 Functional requirement *m*.1 ......
    - 3.2.*m*.*n* Functional requirement *m*.*n*
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organised by Object

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Classes/Objects
  - 3.2.1 Class/Object 1
    - 3.2.1.1 Attributes (direct or inherited)
      - 3.2.1.1.1 Attribute 1 ......
      - 3.2.1.1.*n* Attribute *n*
    - 3.2.1.2 Functions (services, methods, direct or inherited)
      - 3.2.1.2.1 Functional requirement 1.1 ......
      - 3.2.1.2.*m* Functional requirement 1.*m*
    - 3.2.1.3 Messages (communications received or sent)
  - 3.2.2 Class/Object 2 ......
  - 3.2.*p* Class/Object *p*
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organised by Feature

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 System Features
  - 3.2.1 System Feature 1
    - 3.2.1.1 Introduction/Purpose of feature
    - 3.2.1.2 Stimulus/Response sequence
    - 3.2.1.3 Associated functional requirements
      - 3.2.1.3.1 Functional requirement 1 ......
      - 3.2.1.3.n Functional requirement n
  - 3.2.2 System Feature 2 ....
  - 3.2.m System Feature m ......
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organised by Stimulus

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 Stimulus 1
    - 3.2.1.1 Functional requirement 1.1 ......
    - 3.2.1.n Functional requirement 1.n
  - 3.2.2 Stimulus 2 ......
  - 3.2.m Stimulus m
    - 3.2.m.1 Functional requirement m.1 ......
    - 3.2.m.n Functional requirement m.n
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Outline for SRS Section 3
Organised by Response

3 Specific Requirements
 3.1 External interface requirements
  3.1.1 User interfaces
  3.1.2 Hardware interfaces
  3.1.3 Software interfaces
  3.1.4 Communications interfaces
 3.2 Functional requirements
  3.2.1 Response 1
   3.2.1.1 Functional requirement 1.1 ......
   3.2.1.n Functional requirement 1.n
  3.2.2 Response 2 ......
  3.2.m Response m
   3.2.m.1 Functional requirement m.1......
   3.2.m.n Functional requirement m.n
 3.3 Performance Requirements
 3.4 Design Constraints
 3.5 Software system attributes
 3.6 Other requirements

Outline for SRS Section 3
Organised by Functional Hierarchy

3 Specific Requirements
 3.1 External interface requirements
  3.1.1 User interfaces
  3.1.2 Hardware interfaces
  3.1.3 Software interfaces
  3.1.4 Communications interfaces
 3.2 Functional requirements
  3.2.1 Information flows
   3.2.1.1 Data flow diagram 1
    3.2.1.1.1 Data entities
    3.2.1.1.2 Pertinent processes
    3.2.1.1.3 Topology
   3.2.1.2 Data flow diagram 2
    3.2.1.2.1 Data entities
    3.2.1.2.2 Pertinent processes
    3.2.1.2.3 Topology ......
   3.2.1.n Data flow diagram n
    3.2.1.n.1 Data entities
    3.2.1.n.2 Pertinent processes
    3.2.1.n.3 Topology

(Contd)...

3.2.2 Process descriptions
    3.2.2.1 Process 1
        3.2.2.1.1 Input data entities
        3.2.2.1.2 Algorithm or formula of process
        3.2.2.1.3 Affected data entities
    3.2.2.2 Process 2
        3.2.2.2.1 Input data entities
        3.2.2.2.2 Algorithm or formula of process
        3.2.2.2.3 Affected data entities ......
    3.2.2.*m* Process *m*
        3.2.2.*m*.1 Input data entities
        3.2.2.*m*.2 Algorithm or formula of process
        3.2.2.*m*.3 Affected data entities
3.2.3 Data construct specifications
    3.2.3.1 Construct 1
        3.2.3.1.1 Record type
        3.2.3.1.2 Constituent fields
    3.2.3.2 Construct 2
        3.2.3.2.1 Record type
        3.2.3.2.2 Constituent fields ......
    3.2.3.*p* Construct *p*
        3.2.3.*p*.1 Record type
        3.2.3.*p*.2 Constituent fields
3.2.4 Data dictionary
    3.2.4.1 Data element 1
        3.2.4.1.1 Name
        3.2.4.1.2 Representation
        3.2.4.1.3 Units/Format
        3.2.4.1.4 Precision/Accuracy
        3.2.4.1.5 Range
    3.2.4.2 Data element 2
        3.2.4.2.1 Name
        3.2.4.2.2 Representation
        3.2.4.2.3 Units/Format
        3.2.4.2.4 Precision/Accuracy
        3.2.4.2.5 Range ......
    3.2.4.*q* Data element *q*
        3.2.4.*q*.1 Name
        3.2.4.*q*.2 Representation
        3.2.4.*q*.3 Units/Format
        3.2.4.*q*.4 Precision/Accuracy
        3.2.4.*q*.5 Range

3.3 Performance Requirements
3.4 Design Constraints
3.5 Software system attributes
3.6 Other requirements

Outline for SRS Section 3
Showing Multiple Organisations

3 Specific Requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 User class 1
    - 3.2.1.1 Feature 1.1
      - 3.2.1.1.1 Introduction/Purpose of feature
      - 3.2.1.1.2 Stimulus/Response sequence
      - 3.2.1.1.3 Associated functional requirements
    - 3.2.1.2 Feature 1.2
      - 3.2.1.2.1 Introduction/Purpose of feature
      - 3.2.1.2.2 Stimulus/Response sequence
      - 3.2.1.2.3 Associated functional requirements ......
    - 3.2.1.*m* Feature 1.*m*
      - 3.2.1.*m*.1 Introduction/Purpose of feature
      - 3.2.1.*m*.2 Stimulus/Response sequence
      - 3.2.1.*m*.3 Associated functional requirements
  - 3.2.2 User class 2 ......
  - 3.2.*n* User class *n* ......
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software system attributes
- 3.6 Other requirements

3.7 REQUIREMENTS VALIDATION

After the completion of SRS document, we may like to check the document for

- completeness and consistency
- conformance to standards
- requirements conflicts
- technical errors
- ambiguous requirements

The objective of requirements validation is to certify that the SRS document is an acceptable document of the system to be implemented. This helps us to find errors in the document and improves the quality of the software development process. Sometimes we confuse between analysis and validation. However, analysis works with raw requirements as elicited from the various stakeholders whereas validation works with a final draft of the SRS document

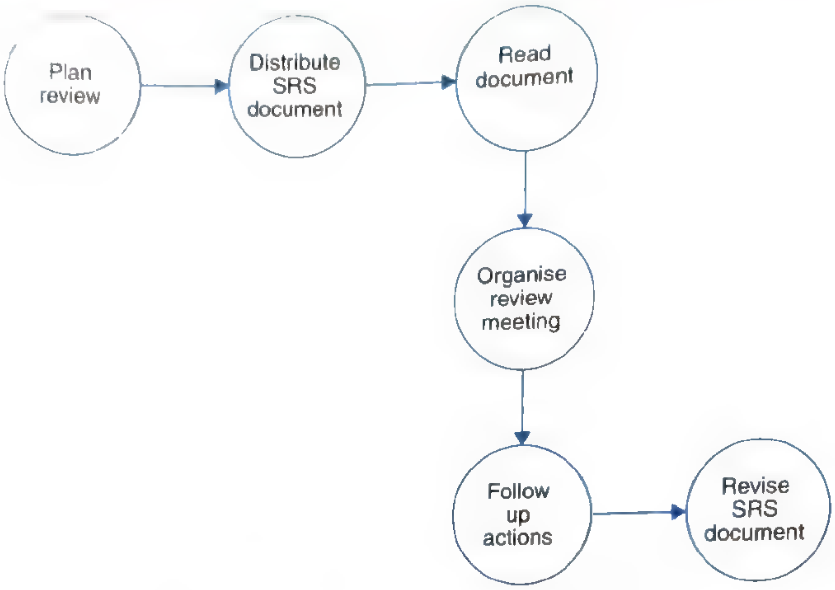


Fig. 3.21: Requirements review process

(*vi*) **Revise SRS document:** The SRS document is revised to reflect the approved actions. At this stage, it may be accepted or may be reviewed.

Problem actions: Some of the problems actions are discussed below :

(*i*) **Requirements clarification:** The requirement may be badly expressed or may have accidentally omitted during the process of elicitation.

(*ii*) **Missing information:** Some information is missing from the SRS document. It is the responsibility of the requirements engineers who are revising the document to find this information from system stakeholders.

(*iii*) **Requirements conflict:** There is a conflict between requirements. The stakeholders involved must negotiate to resolve this conflict.

(*iv*) **Unrealistic requirement:** The requirement does not appear to be implementable with the technology available or within other constraints on the system. Stakeholders must be consulted to decide how to make the requirement more realistic.

(*v*) **Security issues:** Review team may comment on security issues. The security and safety of the system is essential for the survival of the system. The reviewers may review the system in accordance with some security standards.

Membership of review team: The team should involve a number of stakeholders drawn from different backgrounds. People from different backgrounds bring different skills and knowledge to review the SRS document. Stakeholders feel involved in the process and develop an understanding of the needs of other stakeholders. Review team should always involve at least one domain expert and an enduser.

Review checklists

(*i*) **Understandability:** Can readers of the document understand what the requirements mean ?

(*ii*) **Redundancy:** Is information unnecessarily repeated in the SRS document ?

(*iii*) **Completeness:** Does the checker know of any missing requirements or is there any information missing from individual requirement descriptions ?

with negotiated and agreed requirements. The inputs and outputs of validation process are shown in figure 3.20. There are many requirements validation techniques and few are discussed in subsequent subsections.

Fig. 3.20: Validation process with inputs and outputs

SRS document: It should be a final draft, not an unfinished draft and should be formatted/organised as per IEEE standards.

Organisational standards: Every organisation should have some quality standards for SRS document and other activities. These standards will be used for reviewing during validation.

Organisational knowledge: Knowledge, often implicit, of the organisation which may be used to judge the realism of the requirements.

Problem list: List of discovered problems in the requirements document.

Approved actions: List of approved actions in response to the requirements problems. Some problems may have several corrective actions; some problems may have no associated actions.

3.7.1 Requirements Reviews

This is a popular requirements validation technique where a group of people will read the SRS document and look for possible problems. These identified problems may be discussed in the group and some actions may also be approved in order to get rid of these problems. The requirements review process is given in Figure 3.21.

(*i*) **Plan review:** The review team is selected and time and place for review meeting is fixed.

(*ii*) **Distribute SRS document:** The SRS document is distributed to all the members.

(*iii*) **Read SRS document:** Each member should read the document carefully to find conflicts, omissions, inconsistencies, deviations from standards and other problems.

(*iv*) **Organise review meeting:** Each member presents his/her views and identified problems. The problems are discussed and a set of actions to address the problem is approved.

(*v*) **Follow-up actions:** The chairperson of the team checks that the approved actions have been carried out.

(*iv*) **Ambiguity:** Are the requirements expressed using terms which are clearly defined ? Could readers from different backgrounds make different interpretations of the requirements ?

(*v*) **Consistency:** Do the descriptions of different requirements include contradictions ? Are there contradictions between individual requirements and overall system requirements ?

(*vi*) **Organisation:** Is document structured as per standards ? Are the descriptions of requirements organised so that related requirements are grouped ?

(*vii*) **Conformance to standards:** Does the SRS document conform to defined standards ?

(*viii*) **Traceability:** Are requirements unambiguously identified, include links to related requirements and to the reasons why these requirements have been included ?

3.7.2 Prototyping

Prototyping is also used during analysis as explained in section 5.3.4. This is an expensive exercise because an executable model of the system is demonstrated to endusers, customers, and other review team members. Validation prototype should be reasonably complete and efficient and should be used as the required system. Some documents and training may also be necessary to get good results. (For details please refer to section 5.3.4). Prototyping is also a popular validation technique if a prototype has been developed during the requirements elicitation and analysis stage.

3.8 REQUIREMENTS MANAGEMENT

As we have often discussed changes in requirements are inevitable. Many times, we are forced to make changes due to factors beyond our control. Every change is painful and requires some systematic way to handle it. Moreover, after the installation of system, requests for new requirements or changed requirements start and may create challenging situations for the developers. It is difficult to predict what effects the new system will have on the organisation. After the experience of the installed system, customer may find new needs and priorities.

Requirements management is a new term which has been rapidly adopted by industry. It is the process of understanding and controlling changes to system requirements. Most of times, requirements are not independent ; but are dependent on each other. Hence, one change may have implications on other dependent requirements and thus make the task much more difficult and challenging.

3.8.1 Enduring and Volatile Requirements

Requirements are broadly classified in two categories *i.e.*, enduring and volatile.

(*i*) **Enduring requirements:** These are core requirements and are related to main activity of the organisation. For a library management system, issue/return a book, cataloging etc. are core activities and are stable for any system. Hence, enduring requirements are stable and are not changed easily.

(ii) **Volatile requirements.** These requirements are likely to change during software development life cycle or even after delivery of the product. There are many reasons for such changes some of the reasons are :

- Changes to the environment
- Changes in technology
- Changes in policies
- Changes in customer's expectations

3.8.2 Requirements Management Planning

Requirements management planning is very critical, but important for the success of any project. The process of requirement management ends when the final product is released, and the customer is fully satisfied. However, the fewer modifications in requirements, may also be better for everybody. We should be able to trace our requirements all the time. When a customer comes up with an additional issue, it may be too late to change a requirement or add a new one—the workload and costs are simply too large to make it worth it. This remains subject of negotiation between us and the customer; but our task is to know exactly what would be the effect of implementing new requirements, and to translate in to the language of the customer. The customer may not be receptive when he sees how many code lines need to be changed, but he may understand when we tell him how much this will cost.

Tracing requirements also involves additional tests, performed from time to time, to ensure that the process runs smoothly and errors are identified and corrected early on. When faced with a big project, we may have different sets of requirements, some that apply to the entire project, and some for parts of it. When a certain design is implemented for certain requirements, make a note about the effects and the alternatives it may be useful for future projects or even for the same project, if the customer is not satisfied with the result.

3.8.3 Requirements Change Management

The customers expect modern software systems to be evolvable and sufficiently flexible to accomodate changing users needs. This expectation, however, is untempered by the fact that changing requirements are recognised as a major cause of project failure. This may also have a degrading effect on the underlying design of a system. The fact of life is that software requirements change and evolve from inception to deployment.

Hence changes to requirements are carefully traced, analyzed and their effects on the overall system is properly assessed. The requirements change process should include the following activities to be carried out when a change is needed in the requirements .

- Allocating adequate resources (Assignment of responsibilities).
- Analysis of requirement changes (Menagement of changes)
- Documenting requirements (Documentation)
- Creating requirements traces throughout the project life cycle from inception to the final work products (requirements traceability).
- Establishing team communication (communication of change)
- Establishing a baseline for requirements specification (Establishment of baseline).

(*i*) **Assignment of responsibilities:** All changes should be approved or rejected by the competent authority designated for the project. It may be project manager, project lead or any other equivalent person. After the decision, work is further given to individuals to carryout desired modifications.

(*ii*) **Management of changes:** It is an important activity for the implementation of any change. Whenever a request is received for any addition/modification, a change request is created through a specified request form. The request is analyzed due to its impact on overall project cost, resources allocated and delivery schedule of the project. After the implementation of any change, a formal notice is issued for the information to all stakeholders.

(*iii*) **Documentation:** It is required to keep track of every activity of the project. Success of any software project lies in its documentation. It is very pertinent to note that without documentation, project future is in dark and with every passing day, it is leading towards a disaster. Documenting requirements is an iterative process and should be carried out with utmost sincerity.

(*iv*) **Requirements traceability:** It is a technique to provide relationships between requirements, design and implementation in order to manage the effect of change and its impact on the success of the project. Every requirement should be traceable and every piece of design work may be traced to one or more requirements in the project. If we are not able to trace any implementation activity to its requirements, it indicates that either there are potential issues with the accuracy and goodness of the requirements or else the design team is implementing something that is outside the scope of the system.

(*v*) **Communication of change:** The most problematic changes to the requirements specification have been the ones that are not communicated to every stakeholder. Communication failures typically occur when we may drop a feature or change a performance requirement without communication to others. Establishment of automated e-mail notification system may inform everyone in an organised and timely manner. This may only increase the possibility of the success of any change.

(*vi*) **Establishment of baseline:** After the approval of a change, it is communicated to all. If every stakeholder agrees to the change, it is implemented and tested. Baseline is the tested version of a set of requirements representing a conceptual milestone, serves as the basis for further development. A particular version becomes baseline, when a responsible group decides to designate it as such. Baselining is simply labeling a set of requirements at specific versions and freezing them before proceeding to the next phase of development. The steps for baselines establishment are:

- Approval of changes from competent authority.
- Communicate the approval to all stakeholders.
- History of changes is maintained.
- All documents are updated to reflect the changes.

3.9 STUDENT RESULT MANAGEMENT SYSTEM—EXAMPLE

A university has decided to engage a software company for the automation of student result management system of its M. Tech. Programme. The following documents are required to be prepared.

(*i*) Problem statement

(*ii*) Context diagram

(*iii*) Data flow diagrams

(*iv*) ER diagrams

(*v*) Use case diagram

(*vi*) Use cases

(*vii*) SRS as per IEEE std. 830-1993

These seven documents may provide holistic view of the systme to be developed. The SRS will act as contract document between developers (software company) and client (University).

3.9.1 Problem Statement

The problem statement is the first document which is normally prepared by the client. It only, gives superficial view of the system as per client's perspective and expectations. It is the input to the requirement engineering process where final product is the SRS.

The problem statement of student result management system of M. Tech. (Information Technology) Programme of a University is given below:

"A University conducts a 4-semester M. Tech. (IT) program. The students are offered four theory papers and two Lab papers (practicals) during Ist, IInd and IIIrd semesters. The theory papers offered in these semesters are categorized as either 'Core' or 'Elective'. Core papers do not have an alternative subject, whereas elective papers have two other alternative subjects. Thus, a student can study any subject out of the 3 choices available for an elective paper.

In Ist, IInd and IIIrd semesters, 2 core papers and 2 elective papers are offered to each student. The students are also required to submit a term paper/minor project in IInd and IIIrd semesters each. In IVth semester the students have to give a seminar and submit a dissertation on a topic/subject area of their interest.

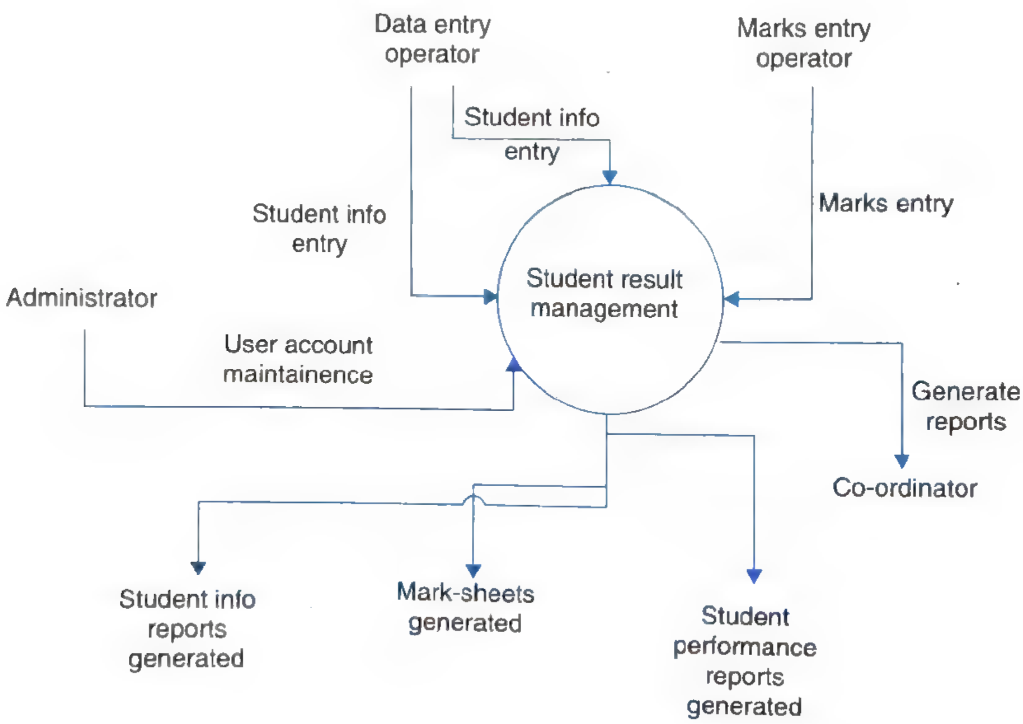
The evaluation of each subject is done out of 100 marks. During the semester, minor exams are conducted for each semester. Students are also required to submit assignments as directed by the corresponding faculty and maintain Lab records for practicals. Based on the students' performance in minor exams, assignments, Lab records and their attendance, marks out of 40 are given in each subject and practical paper. These marks out of 40 account for internal evaluation of the students. At the end of each semester major exams are conducted in each subject (theory as well as practical). These exams are evaluated out of 60 marks and account for external evaluation of the students. Thus, the total marks of a student in a subject are obtained by adding the marks obtained in internal and external evaluation.

Every subject has some credit points assigned to it. If the total marks of a student are > = 50 in a subject, he/she is considered 'Pass' in that subject otherwise the student is considered 'Fail' in that subject. If a student passes in a subject, he/she earns all the credit points assigned to that subject, but if the students fails in a subject he/she does not earn any credit point in that subject. At any time, the latest information about subjects being offered in various semesters and their credit points can be obtained from university's website.

It is required to develop a system that will manage information about subjects offered in various semesters, students enrolled in various semesters, elective (s) opted by various students in different semesters, marks and credit points obtained by students in different semesters. The system should also have the ability to generate printable mark-sheets for each student. Semester-wise detailed mark lists and student performance reports also need to be generated."

3.9.2 Context Diagram

The context diagram is given below:



The following persons are interacting with the "student result management system"

(*i*) Administrator

(*ii*) Marks entry operator

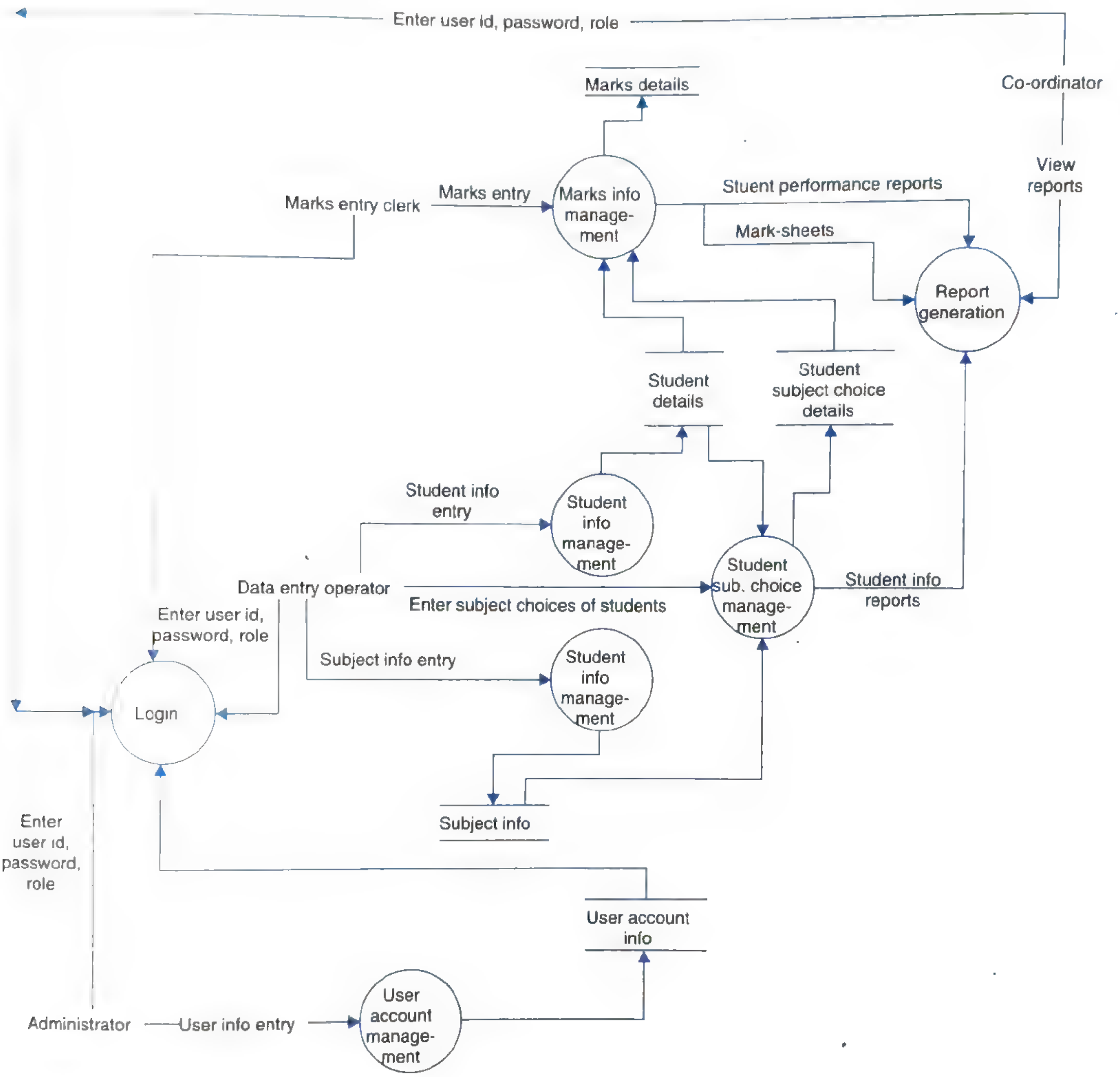
(*iii*) Data entry operator

(*iv*) Co-ordinator

3.9.3 Level-n DFD

Level-1 DFD

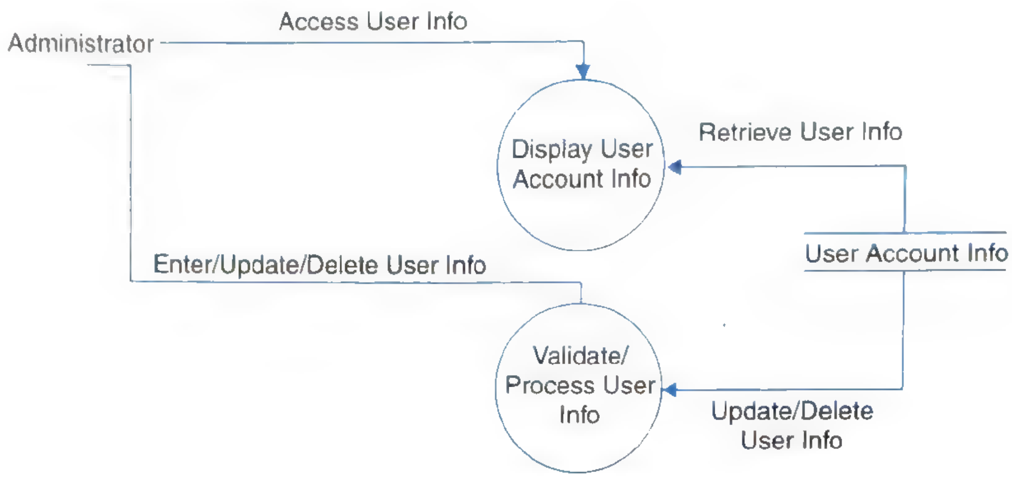
The Level-1 DFD is given below:



Level 1 DFD of result management system

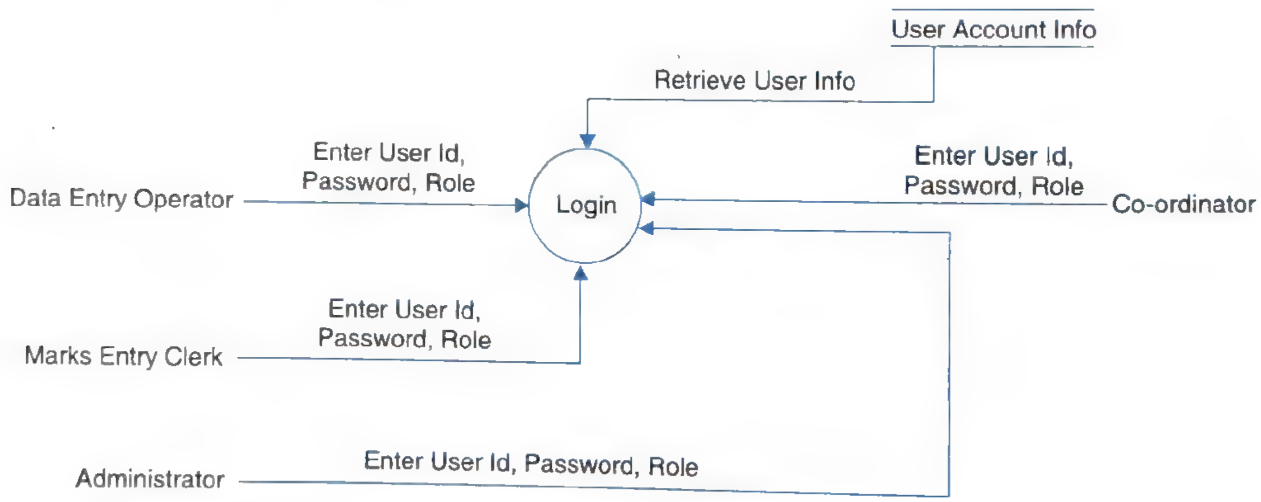
Level-2 DFDs

1. User account maintenance



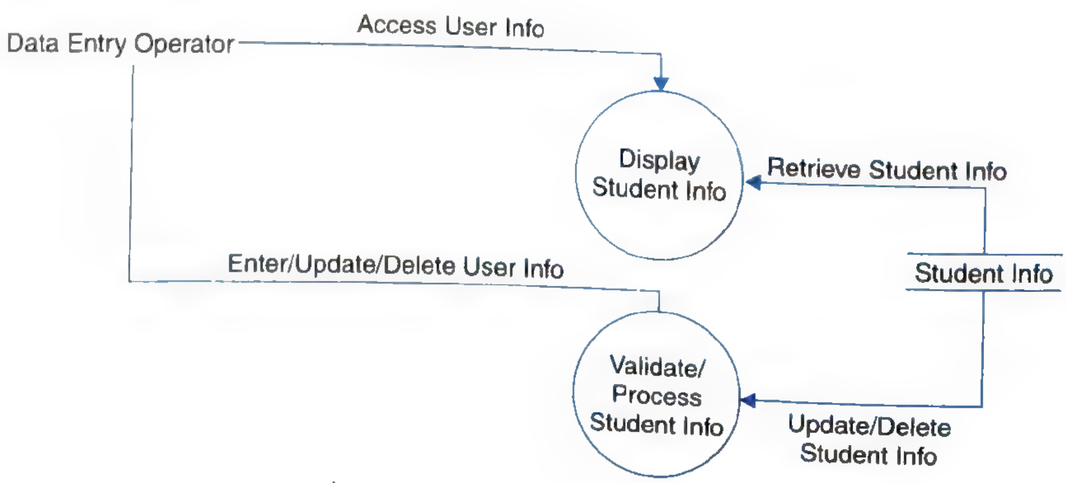
2. Login

The Level 2 DFD of this process is given below:



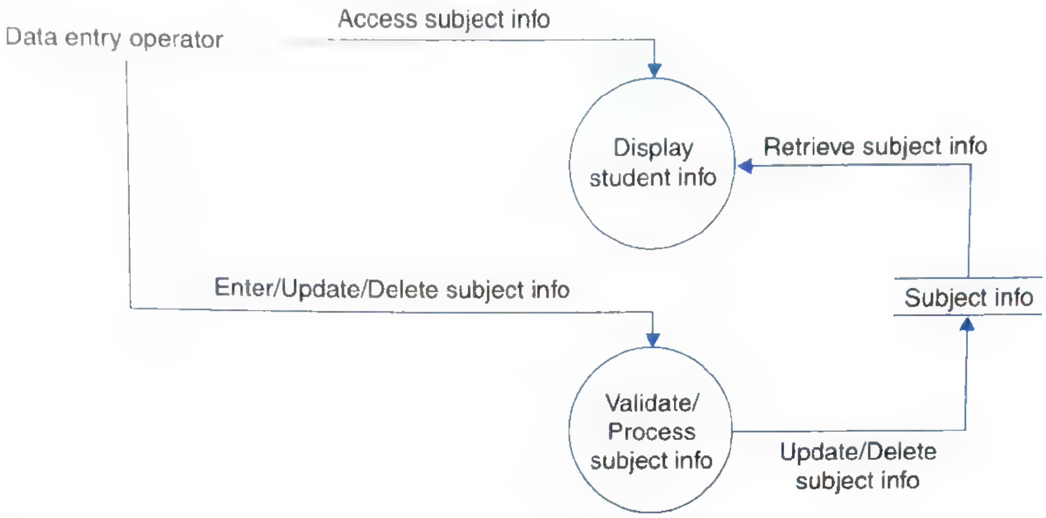
3. Student information management

The Level 2 DFD of this process is given below:



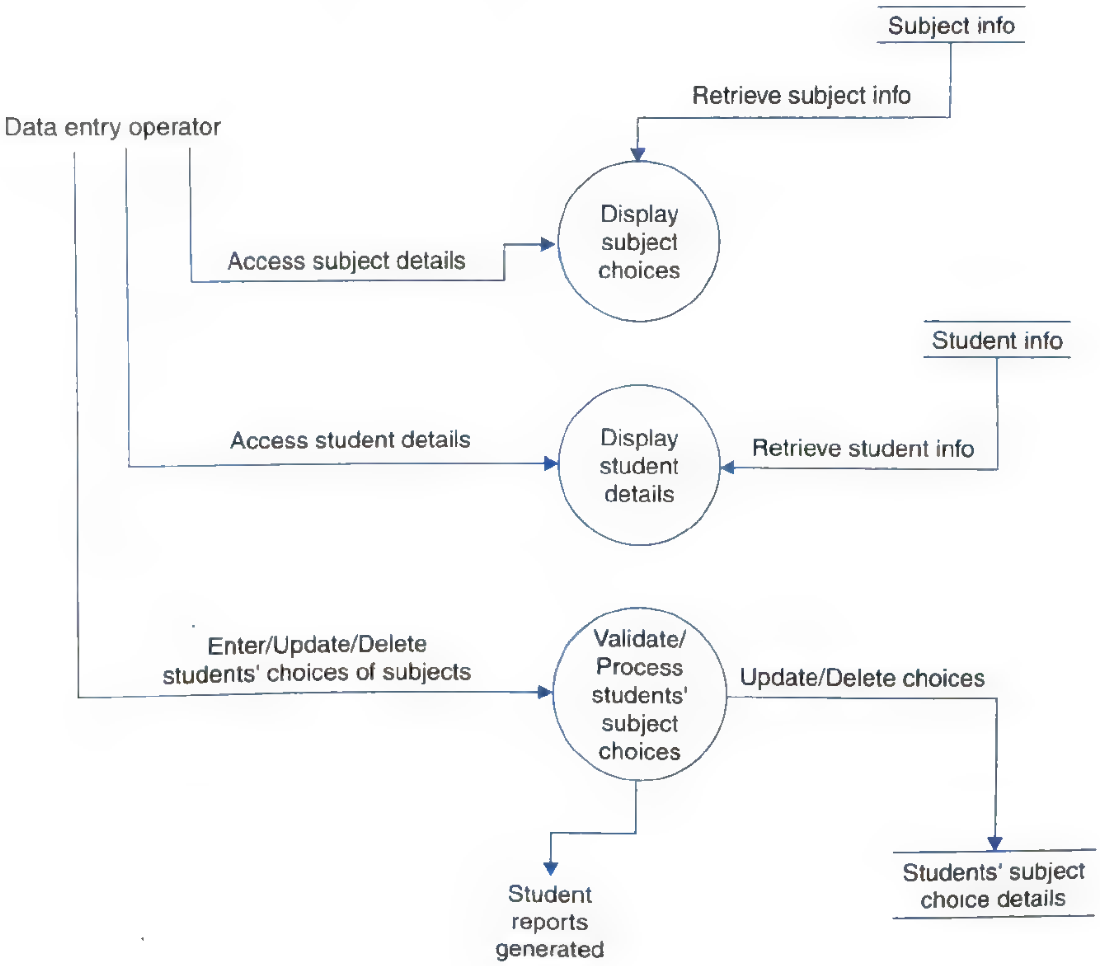
4. Subject information management

The Level 2 DFD of this process is given below:



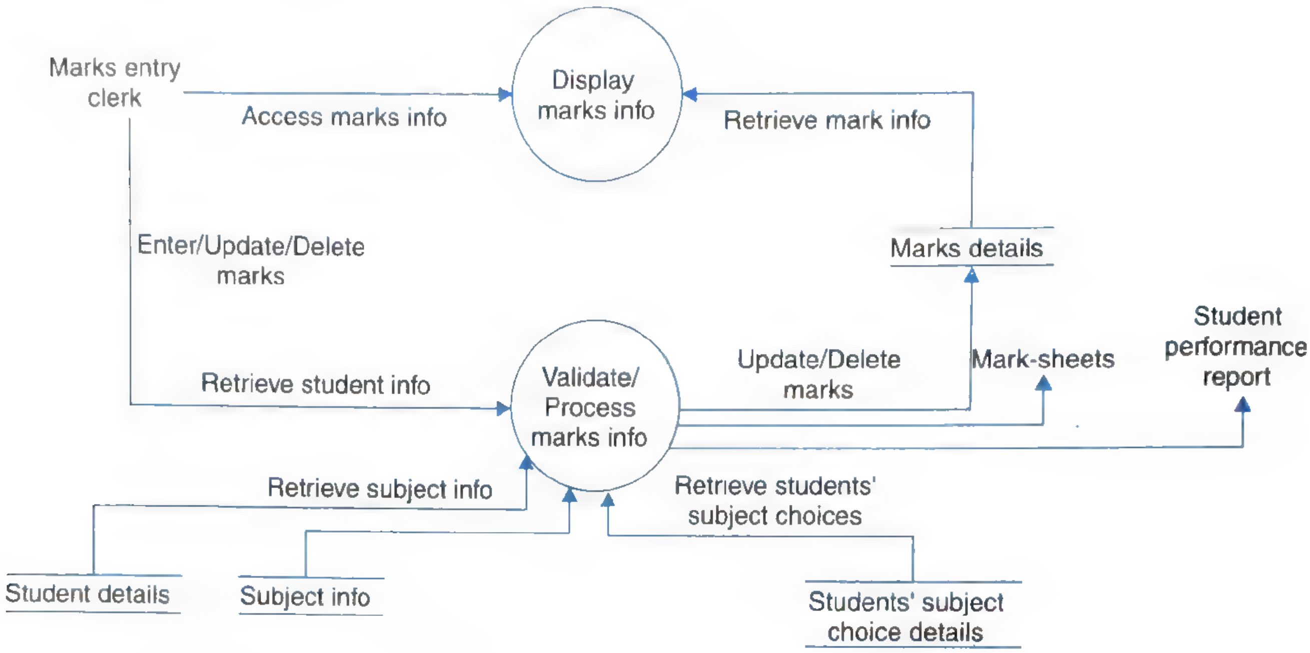
5. Students' subject choice management

The Level 2 DFD of this process is given below:



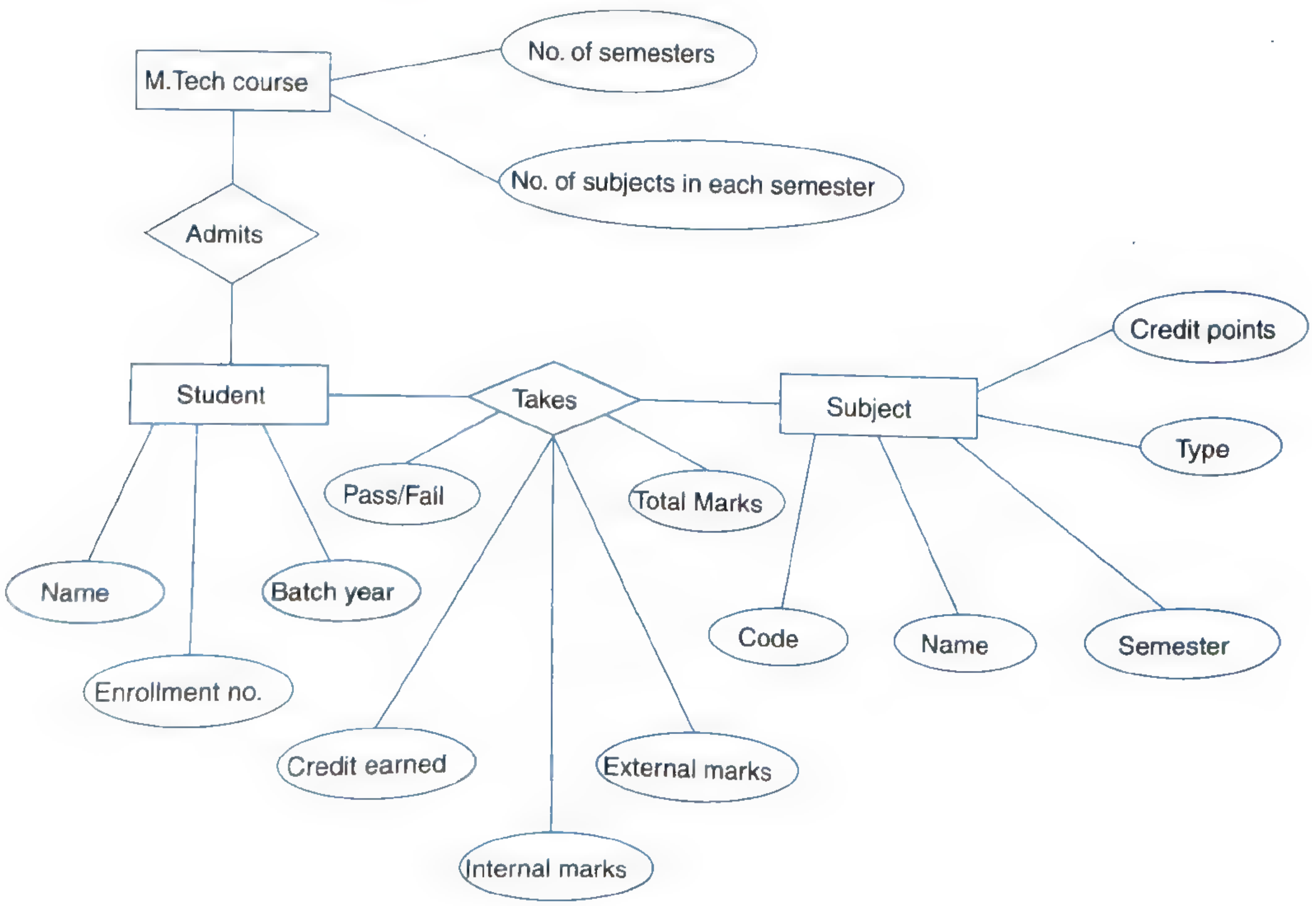
6. Marks information management

The Level 2 DFD of this process is given below:

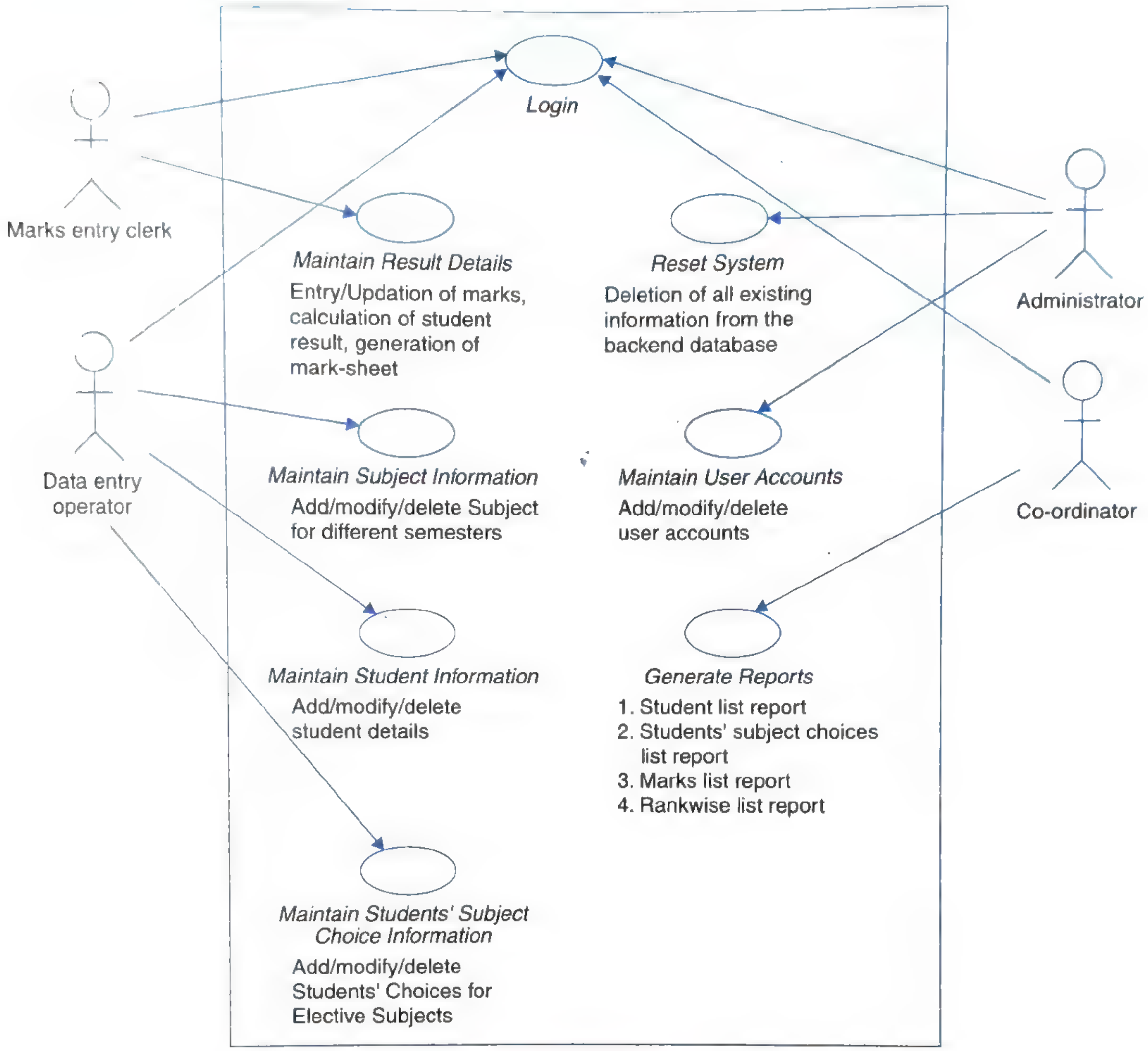


3.9.4 Entity Relationship Diagram

The ER diagram of the system is given below:



3.9.5 Use Case Diagram



Use Case Diagram

3.9.6 Use Cases

1 Login

1.1 Brief Description

This use case describes how a user logs into the Student Result Management System.

1.2 Actors

The following actor(s) interact and participate in this use case:

Data Entry Operator, Marks Entry Clerk, Administrator, Co-ordinator.

1.3 Flow of Events

1.3.1 Basic Flow

This use case starts when the actor wishes to Login to the Student Result Management System.

1. The system requests that the actor enter his/her name, password and role. The role can be any one of Data Entry Operator, Marks Entry Clerk, Co-ordinator, and Administrator.
2. The actor enters his/her name, password and role.
3. The system validates the entered name, password, role and logs the actor into the system.

1.3.2 Alternative Flows

1.3.2.1 Invalid Name/Password/Role

If in the Basic Flow, the actor enters an invalid name, password and/or role, the system displays an error message. The actor can choose to either return to the begining of the Basic Flow or cancel the login, at which point the use case ends.

1.4 Special Requirements

None

1.5 Pre-Conditions

All users must have a User Account (*i.e.*, User ID, Password and Role) created for them in the system (through the Administrator), prior to executing the use cases.

1.6 Post-Conditions

If the use case was successful, the actor is logged into the system. If not, the system state is unchanged.

If the actor has the role 'Data Entry Operator' he/she will have access to only screens corresponding to the Subject Info Maintenance, Student Info Maintenance and Students' Subject Choice Info Maintenance modules of the system.

If the actor has the role 'Marks Entrey Clerk', he/she will have access to only screens corresponding to the Marks Info Maintenance module of the system. If the actor has the role 'Co-ordinator', he/she will only be able to view/print the various reports generated by the system.

If the actor has the role 'Administrator' he/she will have access to only screens corresponding to User Account maintenance module and Reset System feature of the system.

1.7 Extension Points

None

2 Maintain Student Information

2.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain student information. This includes adding, changing and deleting studnet information from the system.

2.2 Actors

The following actor (s) interact and participate in this use case:

Data Entry Operator.

2.3 Flow of Events

2.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to add, change, and/or delete student information from the system.

1. The system requests that the Data Entry Operator specify the function he/she would like to perform (either Add a Student, Update a Student, or Delete a Student).
2. Once the Data Entry Operator provides the requested information, one of the sub-flows is executed.
   - If the Data Entry Operator selected "Add a Student", the **Add a Student** sub-flow is executed.
   - If the Data Entry Operator selected "Update a Student", the **Update a Student** sub-flow is executed.
   - If the Data Entry Operator selected "Delete a Student", the **Delete a Student** sub-flow is executed.

2.3.1.1 Add a Student

1. The system requests that the Data Entry Operator enter the student information. This includes:

   (*a*) Name

   (*b*) Enrollment Number–should be unique for every student

   (*c*) Year of Enrollment
2. Once the Data Entry Operator provides the requested information, the student is added to the system and an appropriate message is displayed.

2.3.1.2 Update a Student

1. The system requests that the Data Entry Operator enters the student enrollment number.
2. The Data Entry Operator enters the student enrollment number. The system retrieves and displays the student information.
3. The Data Entry Operator makes the desired changes to the student information. This includes any of the information specified in the **Add a Student** sub-flow.
4. Once the Data Entry Operator updates the necessary information, the system updates the student record with the updated information.

2.3.1.3 Delete a Student

1. The system requests that the Data Entry Operator enters the student enrollment number.
2. The Data Entry Operator enters the student enrollment number. The system retrieves and displays the student information.
3. The system prompts the Data Entry Operator to confirm the deletion of the student.
4. The Data Entry Operator confirms the deletion.
5. The system deletes the student record.

2.3.2 Alternative Flows

2.3.2.1 Student Not Found

If in the **Update a Student** or **Delete a Student** sub-flows, a student with the specified enrollment number does not exist, the system displays an error message. The Data Entry Operator can then enter a different enrollment number or cancel the operation, at which point the use case ends.

2.3.2.2 Update Cancelled

If in the **Update a Student** sub-flow, the Data Entry Operator decides not to update the student information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

2.3.2.3 Delete Cancelled

If in the **Delete a Student** sub-flow, the Data Entry Operator decides not to delete the student information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

2.4 Special Requirements

None

2.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

2.6 Post-Conditions

If the use case was successful, the student information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

2.7 Extension Points

None

3 Maintain Subject Information

3.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain subject information. This includes adding, changing and deleting subject information from the system.

3.2 Actors

The following actor (s) interact and participate in this use case:

Data Entry Operator

3.3 Flow of Events

3.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to add, change, and/or delete subject information from the system.

1. The system requests that the Data Entry Operator specify the function he/she would like to perform (either Add a Subject, Update a Subject, or Delete a Subject).
2. Once the Data Entry Operator provides the requested information, one of the sub-flows is executed.
   - If the Data Entry Operator selected "Add a Subject", the **Add a Subject** sub-flow is executed.
   - If the Data Entry Operator selected "Update a Subject", the **Update a Subject** sub-flow is executed.
   - If the Data Entry Operator selected "Delete a Subject", the **Delete a Subject** sub-flow is executed.

3.2.1.1 Add a Subject

1. The system requests that the Data Entry Operator enters the subject information. This includes:

   (*a*) Name of the subject

   (*b*) Subject Code–should be unique for every subject

   (*c*) Semester

   (*d*) Subject Type–can be Core 1/Core 2/Dissertation/Elective 1/Elective 2/Lab 1/Lab 2/ Minor Project/Seminar/Term Paper.

   (*e*) Credits.
2. Once the Data Entry Operator provides the requested information, the subject is added to the system and an appropriate message is displayed.

3.3.1.2 Update a Subject

1. The system requests that the Data Entry Operator enters the subject code.
2. The Data Entry Operator enters the subject code. The system retrieves and displays the subject information.
3. The Data Entry Operator makes the desired changes to the subject information. This includes any of the information specified in the **Add a Subject** sub-flow.
4. Once the Data Entry Operator updates the necessary information, the system updates the subject record with the updated information.

3.3.1.3 Delete a Subject

1. The system requests that the Data Entry Operator enter the subject code.
2. The Data Entry Operator enters the subjects code. The system retrieves and displays the subject information.
3. The system prompts the Data Entry operator to confirm the deletion of the subject.
4. The Date Entry Operator confirms the deletion.
5. The system deletes the subject record.

3.3.2 Alternative Flows

3.3.2.1 Subject Not Found

If in the **Update a Subject** or **Delete a Subject** sub-flows, a subject with the specified subject code does not exist, the system displays an error message. The Data Entry Operator can then enter a different subject code or cancel the operation, at which point the use case ends.

3.3.2.2 Update Cancelled

If in the **Update a Subject** sub-flow, the Data Entry Operator decides not to update the subject information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

3.3.2.3 Delete Cancelled

If in the **Delete a Subject** sub-flow, the Data Entry Operator decides not to delete the subject information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

3.4 Special Requirements

None

3.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

3.6 Post–Conditions

If the use case was successful, the student information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

3.7 Extension Points

None

4 Maintain Students' Subject Choice Infromation

4.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain information about the choice of different Elective subjects opted by various students. This includes displaying the various available choices of Elective subjects available during a particular semester and updating the information about the choice of Elective Subject (s) opted by different students of that semester.

4.2 Actors

The following actor (s) interact and participate in this use case:

Data Entry Operator

4.3 Flow of Events

4.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to update students' Subject Choice information from the system.

1. The system requests that the Data Entry Operator specify the semester and enrollment year of students, for which the Students' Subject Choices have to be updated.
2. Once the Data Entry Operator provides the requested information, the system displays the list of available choices for Elective I and Elective II subjects for that semester and the list of students enrolled in the given enrollment year (alongwith their existing subject choices, if any).
3. The system requests that the Data Entry Operator specify the information regarding Students' Subject Choices. this includes .

   (*a*) Student's Enrollment Number

   (*b*) Student's Choice for Elective I subject (the corresponding subject code)

   (*c*) Student's Choice for Elective II subject (the corresponding subject code).
4. Once the Data Entry Operator provides the requested information, the information regarding Student's Subject Choices is added/updated in the system and an appropriate message is displayed.

4.3.2 Alternative Flows

4.3.2.1 Subject Information Does Not Exist

If no or incomplete subject information exists in the system for the semester specified by the Data Entry Operator, the system displays an error message. The Data Entry Operator can then enter a different semester or cancel the operation, at which point the use case ends.

4.3.2.2 Student Information Does Not Exist

If no student information exists in the system for the enrollment year specified by the Data entry Operator, the system displays an error message. The Data Entry Operator can then enter a different enrollment year or cancel the operation, at which point the use case ends.

4.3.2.3 Incorrect Choice Entered for Elective I/Elective II Subjects

If the subject code entered by the Data Entry Operator for Elective I/Elective II subject does not exist in the system, the system displays an error message.

The Data Entry Operator can then enter the correct subject code or cancel the operation, at which point the use case ends.

4.3.2.4 Update Cancelled

If in the **Basic Flow**, the Data Entry Operator decides not to update the subject information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

4.4 Special Requirements

None

4.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

4.6 Post-Conditions

If the use case was successful, information about students' choices for opting different Elective Subjects is added/updated in the system. Otherwise, the system state is unchanged.

4.7 Extension Points

None

5 Maintain Result Details

5.1 Brief Description

This use case allows the actor with role 'Marks Entry Clerk' to maintain subject-wise marks information of each student, in different semesters. This includes adding, changing and deleting marks information from the system.

5.2 Actors

The following actor (s) interact and participate in this use case:
Maks Entry Clerk.

5.3 Flow of Events

5.3.1 Basic Flow

This use case starts when the Marks Entry Clerk wishes to add, change, and/or delete marks information from the system.

1. The system requests that the Marks Entry Clerk specify the function he/she would like to perform (either Add Marks, Update Marks, Delete Marks, or Generate Mark-sheet).
2. Once the Marks Entry Clerk provides the requested information, one of the sub-flows is executed.
   - If the Marks Entry Clerk selected "Add Marks", the **Add Marks** sub-flow is executed.
   - If the Marks Entry Clerk selected "Update Marks", the **Update Marks** sub-flow is executed.
   - If the Marks Entry Clerk selected "Delete Marks", the **Delete Marks** sub-flow is executed.
   - If the Marks Entry Clerk selected "Generate Mark-sheet", the **Generate Mark-sheet** sub-flow is executed.

5.3.1.1 Add Marks Record

1. The system requests that the Marks Entry Clerk enters the marks information. This includes:

(*a*) Selecting a semester

(*b*) Selecting a subject code

(*c*) Selecting the student enrollment number

(*d*) Entering the internal/external marks for that semester, subject code and enrollment number.

2. Once the Marks Entry Clerk provides the requested information, the system saves the marks and an appropriate message is displayed.

5.3.1.2 Update Marks Record

1. The system requests the Marks Entry Clerk to make following entries:

   (*a*) Selecting the semester

   (*b*) Selecting the subject code for which marks have to be updated

   (*c*) Selecting the student enrollment number.

2. Once the Marks Entry Clerk provides the requested information, the system retrieves and displays the corresponding marks details.
3. The Marks Entry Clerk makes the desired changes to the internal/external marks details.
4. The system updates the marks record with the changed information.

5.3.1.3 Delete Marks Record

1. The system requests the Marks Entry Clerk to make following entries:

   (*a*) Selecting the semester

   (*b*) Selecting the subject code for which marks have to be updated

   (*c*) Selecting the student enrollment number.

2. Once the Marks Entry Clerk provides the requested information, the system retrievs and displays the corresponding marks record from the database.
3. The system verifies if the Marks Entry Clerk wishes to proceed with the deletion of the record. Upon confirmation, the record is deleted from the system.

5.3.1.4 Compute Result

1. Once all the marks are added to the database, the result is computed for each student.
2. If the student has scored more than 50% in a subject, the associated credit points are allotted to that student.
3. The average percentage marks are claculated for the student and his/her division is also derived based on the percentage.

5.3.1.5 Generate Mark-Sheet

1. The system requests that the Marks Entry Clerk specify the Enrollment Number of the student and the semester for which mark-sheet is to be generated.
2. Once the Marks Entry Clerk provides the requested information, the system generates a printable mark-sheet for the specified student and displays it.
3. The Marks Entry Clerk can then issue a print request for the mark-sheet to be printed.

5.3.2 Alternative Flows

5.3.2.1 Record Not Found

If in the **Update Marks, Delete Marks** or **Generate Mark-sheet** sub-flows, a record with the specified information does not exist, the system displays an error message. The Marks Entry Clerk can then enter different information for retrieving the record or cancel the operation, at which point the use case ends.

5.3.2.2 Update Cancelled

If in the **Update Marks** sub-flow, the Marks Entry Clerk decides not to update the marks, the update is cancelled and the **Basic Flow** is re-started at the beginning.

5.3.2.3 Delete Cancelled

If in the **Delete Marks** sub-flow, the Marks Entry Clerk decides not to delete the marks, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

5.4 Special Requirements

None

5.5 Pre-Conditions

The Marks Entry Clerk must be logged onto the system before this use case begins.

5.6 Post–Conditions

If the use case was successful, the marks information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

5.7 Extension Points

None

6 Generate Reports

6.1 Brief Description

This use case allows the actor with role 'Co-ordinator' to generate various reports. The following reports can be generated:

(*a*) Student List Report

(*b*) Students' Subject Choices List Report

(*c*) Marks List Report

(*d*) Rank-wise List Report

6.2 Actors

The following actor(s) interact and participate in this use case:

Co-ordinator

6.3 Flow of Events

6.3.1 Basic Flow

This use case starts when the Co-ordinator wishes to generate reports.

1. The system requests the Co-ordinator specify the report he/she would like to generate.
2. Once the Co-ordinator provides the requested information, one of the sub-flows is executed:
   - If the Co-ordinator selected "Student List Report", the **Generate Student List Report** sub-flow is executed.
   - If the Co-ordinator selected "Students' Subject Choices List Report", the **Generate Students' Subject Choices List Report** sub-flow is executed.
   - If the Co-ordinator selected "Marks List Report", the **Generate Marks List Report** sub-flow is executed.
   - If the Co-ordinator selected "Rank-wise List Report", the **Generate Rank-wise List Report** sub-flow is executed.

6.3.1.1 Generate Student List Report

1. The system requests that the Co-ordinator provide the enrollment year for which the Student List report is to be generated.
2. Once the Co-ordinator provides the requested infromation, the system generates the Student List report, containing the list of students enrolled in the given year.
3. The Co-ordinator can then issue a print request for the report to be printed.

6.3.1.2 Generate Student's Subject Choices List Report

1. The system requests that the Co-ordinator provides the enrollment year and the semester for which the Students' Subject Choices List report is to be generated.
2. Once the Co-ordinator provides the requested information, the system generates the Students' subject Choices List report, containing the choices for Elective I and Elective II subjects, opted by the students of the given enrollment year and semester.
3. The Co-ordinator can then issue a print request for the report to be printed.

6.3.1.3 Generate Marks List Report

1. The system requests that the Co-ordinator provides the enrollment year and the semester for which the Marks List report is to be generated.
2. Once the Co-ordinator provides the requested information, the system generates the Marks List report, containing the marks details of various students in all the subjects for the given enrollment year and semester.
3. The Co-ordinator can then issue a print request for the report to be printed.

6.3.1.4 Generate Rank-wise List Report

1. The system requests that the Co-ordinator provide the enrollment year and the semester for which the Rank-wise List report is to be generated.

2. Once the Co-ordinator provides the requested information, the system generates the Rank-wise List report, containing the percentage wise and rank-wise list of all students (alongwith their total marks and division) for the given enrollment year and semester.
3. The Co-ordinator can then issue a print request for the report to be printed.

6.3.2 *Alternative Flows*

6.3.2.1 Student Not Found

If no student information exists in the system for the enrollment year specified by the Co-ordinator, the system displays an error message. The Co-ordinator can then enter a different enrollment year or cancel the operation, at which point the use case ends.

6.4 Special Requirements

None

6.5 Pre-Conditions

The Co-ordinator must be logged onto the system before this use case begins.

6.6 Post–Conditions

If the use case was successful, the desired report is generated. Otherwise, the system state is unchanged.

6.7 Extension Points

None.

7 Maintain User Accounts

7.1 Brief Description

This use case allows the actor with role 'Administrator' to maintain User Account. This includes adding, changing and deleting user account information from the system.

7.2 Actors

The following actor (s) interact and participate in this use case:
Administrator.

7.3 Flow of Events

7.3.1 Basic Flow

This use case starts when the Administrator wishes to add, change, and/or delete use account information from the system.

1. The system requests that the Administrator specify the function he/she would like to perform (either Add a User Account, Update a User Account, or Delete a User Account).
2. Once the Administrator provides the requested information, one of the sub-flows is executed.

- If the Administrator selected "Add a User Account", the **Add a User Account** sub-flow is executed.
- If the Administrator selected "Update a User Account", the **Update a User Account** sub-flow is executed.
- If the Administrator selected "Delete a User Account", the **Delete a User Account** sub-flow is executed.

7.3.1.1 Add a User Account

1. The system requests that the Administrator enters the user information. This includes:
   (a) User Name
   (b) User ID-should be unique for each user account
   (c) Password
   (d) Role
2. Once the Administrator provides the requested information, the user account information is added to the system and an appropriate message is displayed.

7.3.1.2 Update a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The Adiministrator makes the desired changes to the user account information. This includes any of the information specified in the **Add a User Account** sub-flow.
4. Once the Administrator updates the necessary information, the system updates the user account record with the updated information.

7.3.1.3 Delete a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The system prompts the Administrator to confirm the deletion of the user account.
4. The Administrator confirms the deletion.
5. The system deletes the user account record.

7.3.2 Alternative Flows

7.3.2.1 User Not Found

If in the **Update a User Account** or **Delete a User Account** sub-flows, a user account with the specified User ID does not exist, the system displays an error message. The Administrator can then enter a different User ID or cancel the operation, at which point the use case ends.

7.3.2.2 Update Cancelled

If in the **Update a User Account** sub-flow, the Administrator decides not to update the user account information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

7.3.2.3 Delete Cancelled

If in the **Delete a User Account** sub-flow, the Administrator decides not to delete the user account information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

7.4 Special Rcquirements

None

7.5 Pre-Conditions

The Administrator must be logged onto the system before this use case begins.

7.6 Post-Conditions

If the use case was successful, the user account information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

7.7 Extension Points

None

8. Reset System

8.1 Brief Description

This Use case allows the actor with role 'Administrator' to reset the system by deleting all existing information from the system.

8.2 Actors

The following actor (s) interact and participate in this use case:
Adiministrator

8.3 Flow of Events

8.3.1 Basic Flow

This use case starts when the Administrator wishes to reset the system.

1. The system requests the Administrator to confirm if he/she wants to delete all the existing information from the system.
2. Once the Administrator provides confirmation, the system deletes all the existing information from the backend database and displays an appropriate message.

8.3.2 Alternative Flows

8.3.2.1 Reset Cancelled

If in the **Basic Flow,** the Administrator decides not to delete the entire existing information, the reset is cancelled and the use case ends.

8.4 Special Requirements

None

8.5 Pre-Conditions

The Administrator must be logged onto the system before this use case begins.

8.6 Post-Conditions

If the use case was successful, all the existing information is deleted from the backend database of the system. Otherwise, the system state is unchanged.

8.7 Extension Points

None

3.9.7 SRS Document

1 Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, Acronyms, and Abbreviations
- 1.4 References
- 1.5 Overview

2 Overall Description
- 2.1 Product Perspective
  - 2.1.1 System Interfaces
  - 2.1.2 User Interfaces
  - 2.1.3 Hardware Interfaces
  - 2.1.4 Software Interfaces
  - 2.1.5 Communications Interfaces
  - 2.1.6 Memory Constraints
  - 2.1.7 Operations
  - 2.1.8 Site Adaptation Requirements
- 2.2 Product Functions
- 2.3 User Characteristics
- 2.4 Constraints
- 2.5 Assumptions and Dependencies
- 2.6 Apportioning of Requirements

3 Specific Requirements
- 3.1 External Interface Requirements
  - 3.1.1 User Interfaces
    - Login Screen:
    - Subject Info Parameters Screen:
    - Subject Information Screen:
    - Student Info Parameters Screen:
    - Student Information Screen:
    - Students' Subject Choice Parameters Screen:
    - Students' Subject Choice Information Screen:

(Contd.)...

1 Introduction

This document aims at defining the overall software requirements for 'Student Result Management System'. Efforts have been made to define the requirements exhaustively and

accurately. The final product will be having only features/functionalities mentioned in this document and assumptions for any additional functionality/feature should not be made by any of the parties involved in developing/testing/implementing/using this product. In case it is required to have some additional features, a formal change request will need to be raised and subsequently a new release of this document and/or product will be produced.

1.1 Purpose

This specification document describes the capabilities that will be provided by the software application 'Student Result Management System'. It also states the various required constraints by which the system will abide. The intended audience for this document are the development team, testing team and end users of the product.

1.2 Scope

The software product 'Student Result Management System' will be an MIS and Reporting application that will be used for result preparation and management of M. Tech. Program of a University. The application will manage the information about various students enrolled in this course in different years, the subjects offered during different semesters of the course, the students' choices for opting different subjects, and the marks obtained by various students in various subjects in different semesters. Printable reports regarding list of students, marks obtained by all students in a particular semester and performance of students (rank-wise, percentage-wise, pass/fail, division-wise.) will be generated. The system will also generate printable mark-sheets for individual students.

The application will greatly simplify and speed up the result preparation and management process.

1.3 Definitions, Acronyms, and Abbreviations

Following abbreviations have been used throughout this document:

M.Tech: Master of Technology

IT: Information Technology

DBA: Database Administrator

1.4 References

(*i*) *University website:* For information about course structure of M.Tech. Program

(*ii*) IEEE Recommended Practice for Software Requirements Specifications–IEEE Std 830-1993

1.5 Overview

The rest of this SRS document describes the various system requirements, interfaces, features and functionalities in detail.

2 Overall Description

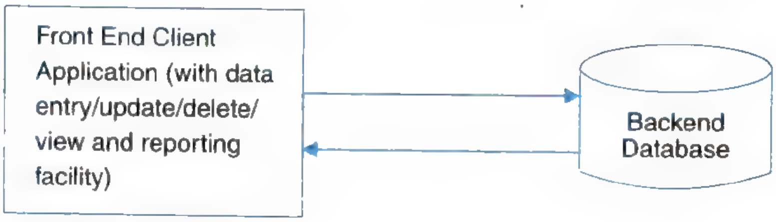
M.Tech. Program is a 4-semester course. The students are offered 4 subjects (theory) and 2 Labs (practical) during first, second and third semesters. Students also have to submit a term paper/minor project in 2nd and 3rd semesters. The fourth semester consists of a seminar and

disseratation. Each subject/lab/term paper/seminar/dissertation has credits associated with it. When a student secures pass marks in a paper he/she also earns all the credit (s) assigned to that paper.

The 'Student Result Management System' will have capability to maintain information about students enrolled in the course, the subjects offered to students during different semesters, the students' choices for opting different Elective subjects (out of the available ones) and the marks obtained by students in different subjects in various semesters. The software will also generate summary reports regarding student information, semester-wise mark lists and performance reports. Printable mark-sheets of individual students will also be generated by the application.

2.1 Product Perspective

The application will be a windows-based, self-contained and independent software product.



2.1.1 System Interfaces

None

2.1.2 User Interfaces

The application will have a user-friendly and menu based interface. Following screens will be provided:

(*i*) A Login screen for entering the username, password and role (Administrator, Data Entry Operator, Marks Entry Clerk, Co-ordinator) will be provided. Access to different screens will be based upon the role of the user.

(*ii*) There will be a screen for capturing and displaying information regarding what all subjects are offered during which semester, how many credit points are assigned to that subject and whether the subject is an elective, a core paper, a lab paper, a term paper or a dissertation.

(*iii*) There will be a screen for capturing and displaying information regarding various students enrolled for the course in different years.

(*iv*) There will be a screen for capturing and displaying information regarding which student is currently enrolled in which semester and what all elective subjects he/she has opted.

(*v*) There will be a screen that will capture information regarding which student has scored how many marks (internal + external evaluation) in each subject (in a particular semester). Credits in each subject will be calculated depending upon the marks obtained in that subject.

(*vi*) There will be a screen for capturing and displaying information regarding which all user accounts exist in the system, thus showing who all can access the system.

The following reports will be generated:

(*i*) *Students' List Report:* Printable reports will be generated to show the list of students enrolled in a particular batch year.

(*ii*) *Students' Subject Choices List Report:* For Ist, IInd and IIIrd semester, there will be printable reports showing the different elective subjects opted by various students (enrolled in a particular batch year) of the corresponding semester.

(*iii*) *Marks List Report:* For each semester there will be a printable report showing the subject-wise marks details for all students of that semester.

(*iv*) *Rank-wise List Report:* For each semester there will be a printable report showing the percentage-wise and rank-wise list of students alongwith the division secured.

(*v*) *Mark-sheet:* For each student of each semester, a printable mark-sheet will be generated, showing the subject-wise marks details, Total marks, total credits, Percentage, Pass/Fail status for that student.

2.1.3 Hardware Interfaces

(*i*) Screen resolution of at least 800 × 600–required for proper and complete viewing of screens. Higher resolution would not be a problem.

(*ii*) Support for printer (dot-matrix/deskJet/inkjet etc.–any will do)–that is, appropriate drivers are installed and printer connected printer will be required for printing of reports and mark-sheets.

(*iii*) Standalone system or network based–not a concern, as it will be possible to run the application on any of these.

2.1.4 Software Interfaces

(*i*) Any windows-based operating system (Windows 95/98/2000/XP/NT)

(*ii*) MS Access 2000 as the DBMS—for database. Future release of the application will aim at upgrading to Oracle 8*i* as the DBMS.

(*iii*) Crystal Reports 8—for generating and viewing reports.

(*iv*) Visual Basic 6—for coding/developing the software.

Software mentioned in pts. (*iii*) and (*iv*) above, will be required only for development of the application. The final application will be packaged as an independent setup program that will be delivered to the client (University in this case).

2.1.5 Communications Interfaces

None

2.1.6 Memory Constraints

At least 64 MB RAM and 2 GB space on hard disk will be required for running the application.

2.1.7 Operations

This product release will not cover any automated housekeeping aspects of the database. The DBA at the client site (*i.e.*, University) will be responsible for manually deleting old/non-required data. Database backup and recovery will also have to be handled by the DBA.

However, the system will provide a 'RESET SYSTEM' function that will delete (upon confirmation from the Administrator) all the existing information from the database.

2.1.8 Site Adaptation Requirements

The terminals at client site will have to support the hardware and software interfaces specified in above sections.

2.2 Product Functions

The system will allow access only to authorised users with specific roles (Administrator, Data Entry Operator, Marks Entry Clerk and Co-ordinator). Depending upon the user's role, he/she will be able to access only specific modules of the system.

A summary of the major functions that the software will perform:

(*i*) A Login facility for enabling only authorised access to the system.

(*ii*) User (with role Data Entry Operator) will be able to add/modify/delete information about different students that are enrolled for the course in different years.

(*iii*) User (with role Data Entry Operator) will be able to add/modify/delete information about different subjects that are offered in a particular semester. The semester-wise list of subjects along with their credit points and type (*i.e.*, elective/core/lab/term paper/dissertation) will also be displayed.

(*iv*) User (with role Data Entry Operator) will be able to add/modify/delete information about the Elective subjects opted by different students in different semesters.

(*v*) User (with role Marks Entry Clerk) will be able to add/modify/delete information regarding marks obtained by different students in different semesters.

(*vi*) User (with role Marks Entry Clerk) will also be able to print mark-sheets of students.

(*vii*) User (with role Co-ordinator) will be able to generate Printable reports (as mentioned in section 2.1.2 above).

(*viii*) User (with role Administrator) will be able to 'Reset' the system–leading to deletion of all existing information from the backend database.

(*ix*) User (with role Administrator) will be able to create/modify/delete new/existing user accounts.

2.3 User Characteristics

- *Educational level:* At least graduate should be comfortable with English language.
- *Experience:* Should be well versed/informed about the course structure of M. Tech. program of University. Entry of marks or their modification can be done only by user who is authorised for this job by the result preparation committee of University.

- *Technical expertise:* Should be comfortable using general-purpose applications on a computer.

2.4 Constraints

(*i*) Since the DBMS being used is MS Access 2000, which is not a very powerful DBMS, it will not be able to store a very huge number of records.

(*ii*) Due to limited features of DBMS being used performance tuning features will not be applied to the queries and thus the system may become slow with the increase in number of records being stored.

(*iii*) Due to limited features of DBMS being used, database auditing will also not be provided.

(*iv*) Users at University will have to implement a security policy to safeguard the marks-related information from being modified by unauthorised users (by means of gaining access to the backend database).

2.5 Assumptions and Dependencies

(*i*) The number of subjects to be taken up by a student in each semester does not change.

(*ii*) The subject types (*i.e.*, elective, core, lab, term paper and dissertation) do not change.

(*iii*) The number of semesters in the M. Tech. Program does not change.

2.6 Apportioning of Requirements

Not Required.

3 Specific Requirements

This section contains the software requirements to a level of detail sufficient to enable designers to design the system, and testers to test that system.

3.1 External Interface Requirements

3.1.1 User Interfaces

The following screens will be provided:

Login screen:

This will be the first screen that will be displayed. It will allow user to access different screens based upon the user's role. Various fields available on this screen will be

(*i*) *User ID:* Alphanumeric of length upto 10 characters

(*ii*) *Password:* Alphanumeric of length upto 8 characters

(*iii*) Role: Will have the following values:

Administrator, Marks Entry Clerk, Co-ordinator, Data Entry Operator

Subject info parameters screen:

This screen will be accessible only to user with role Administrator. It will allow the user to enter the semester number for which the user wants to access the subject information.

Subject information screen:

This screen will be accessible only to user with role Administrator. It will allow user to add, modify/delete information about new/existing subject (s) for the semester that was selected in the 'Subject Info Parameters' screen. The list of available subjects for that semester, will also be displayed. Various fields available on this screen will be:

(i) *Subject Code:* of the format IT-### (# represents a digit)

(ii) *Subject Name:* Alphanumeric, of length upto 50 characters

(iii) *Category / Type:* Will have any of the following values:-

Elective 1/Elective 2/Core/Lab/Term paper/Seminar/Dissertation

(iv) *Credits:* Numeric, will have any value from 0 to 20.

Student info parameters screen:

This screen will be accessible only to user with role Administrator. It will allow the user to enter the Batch Year for which the user wants to access the student information.

Student information screen:

This screen will be accessible only to user role Administrator. It will allow the user to add modify/delete information about new/existing student (s) for a particular Batch Year. Batch Year-wise list of students will also be displayed. Various fields available on these screens will be:

(i) *Student Enrollment No:* of the format ##/M.Tech. (IT)/YYYY (# represents a digit and YYYY represents the batch year).

(ii) *Student Name:* will have only alphabetic letters and length upto 40 characters.

(iii) *Batch Year:* of the format YYYY (representing the year in which the student enrolled for the course).

Students' subject choice parameters screen:

This screen will be accessible only to user with role Administrator. It will allow the user to enter the Batch Year and the semester number for which the user wants to access the students' subject choice information.

Students' subject choice information screen:

This screen will be accessible only to user with role Administrator. It will allow user to add modify/delete students' choices for elective subjects of the semester and batch year selected in "Students' Subject Choice Parameters" screen. For the selected semester it will display the list of available choices for Elective I and for Elective II. The screen will display the list of students enrolled during the selected batch year and currently studying in the selected semester and the user will be able to view/add/modify/delete the subject choices for each student in the list.

Marks entry parameters screen:

This screen will be accessible only to user with role Marks Entry Clerk. It will allow the user to enter the Batch Year, the semester number and the Subject for which the user wants to access the marks information.

Marks entry screen:

This screen will be accessible only to user with role Marks Entry Clerk. It will allow user to add/modify/delete information about marks obtained in the selected subjects by different students of that semester who were enrolled in the Batch Year selected in the 'Marks Entry Parameters' Screen. The screen will display the list of students enrolled during the selected batch year and currently studying the selected subject in the selected semester and the user will be able to view/add/modify/delete the marks for each student in the list.

Various fields available on this screens will be:

(i) *Student Enrollment No.:* will display the enrollment numbers of all students of the selected Batch Year studying the selected subject in the selected semester.

(ii) *Student Name:* will display the name of the student

(iii) *Internal Marks:* between 0 and 40

(iv) *External Marks:* between 0 and 60

(v) *Total Marks:* sum of Internal Marks and External Marks

Mark-sheet parameters screen:

This screen will be accessible only to user with role Marks Entry Clerk. It will allow the user to enter the Enrollment Number and the semester number of the student for whom the user wants to view/print the mark-sheet.

Students list report parameters screen:

This screen will be accessible only to user with role Co-ordinator. It will allow the user to enter the Batch Year for which the user wants to view/print the students list report.

Marks list report parameters screen:

This screen will be accessible only to user with role Co-ordinator. It will allow the user to enter the Batch Year and the semester for which the user wants to view/print the marks list report.

Rank-wise list report parameters screen:

This screen will be accessible only to user with role Co-ordinator. It will allow the user to enter the Batch Year and the semester for which the user wants to view/print the rank-wise list report.

Students' subject choices list report parameters screen:

This screen will be accessible only to user with role Co-ordinator. It will allow the user to enter the Batch Year and the semester for which the user wants to view/print the students' subject choices list report.

3.1.2 Hardware Interfaces

As stated in Section 2.1.3.

3.1.3 Software Interfaces

As stated in section 2.1.4.

3.1.4 Communications Interfaces

None

3.2 System Features

3.2.1 Subject Information Maintenance

Description

The system will maintain information about various subjects being offered during different semesters of the course. The following information would be maintained for each subject:

Subject code, Subject name, Subject Type (Core/Elective 1/Elective 2/Lab 1/Lab 2/Term Paper Minor Project/Dissertation/Seminar), Semester, Credits.

The system will allow creation/modification/deletion of new/existing subjects and also have the ability to list all the available subjects for a particular semester.

Validity checks

(*i*) Only user with role Data Entry Operator will be authorised to access the Subject Information Maintenance module.

(*ii*) Ist, IInd and IIIrd semesters will have 2 core papers, 2 Elective papers, 2 Lab papers and 1 term paper/Minor Project.

(*iii*) Ist, IInd and IIIrd semesters will have 3 choices (subjects) each of type Elective 1 and of type Elective 2.

(*iv*) IVth semester will have only 1 dissertation and 1 seminar.

(*v*) No two semesters will have the same subject *i.e.*, A subject will be offered only in a particular semester.

(*vi*) Subject code will be unique for every subject.

(*vii*) subject code cannot be blank.

(*viii*) Subject name cannot be blank.

(*ix*) Credits cannot be blank.

(*x*) Credits can have value only between 0 and 20.

(*xi*) Subject Type cannot be blank.

(*xii*) Semester cannot be blank.

Sequencing information

Subject info for a particular semester will have to be entered in the system before any student marks information for that semester can be entered.

Error handling/response to abnormal situations

If any of the above validations/sequencing flow does not hold true, appropriate error messages will be prompted to the user for doing the needful.

3.2.2 Student Information Maintenance

Description

The system will maintain information about choice of different Elective subjects opted by various students of different enrollment years in different semesters. The following information would be maintained:

Student Enrollment number, Semester, Student's Choice for Elective 1 subject Student's Choice for Elective 2 subject.

The system will allow creation/modification/deletion of new/existing students and also have the ability to list all the students enrolled in a particular year.

Validity checks

(*i*) Only user with role Data Entry Operator will be authorised to access the Student Information Maintenance module.

(*ii*) Every student will have a unique Enrollment Number.

(*iii*) Enrollment Number cannot be blank.

(*iv*) Student name cannot be blank.

(*v*) Enrollment Year cannot be blank.

Sequencing information

Student Info for a particular student will have to be entered in the system before any marks info can be entered for that student.

Error handling/response to abnormal situations

If any of the above validations/sequencing flow does not hold true, appropriate error messages will be prompted to the user for doing the needful.

3.2.3 Students' Subject Choices Information Maintenance

Description

The system will maintain information about choice of different Elective subjects opted by various students of different enrollment years in different semesters. The following information would be maintained:

Student Enrollment number, Semester, Student's Choice for Elective 1 subject, Student's Choice for Elective 2 subject.

The system will allow creation/modification/deletion of students' subject choices and also have the ability to list all the available students' subject choices for a particular semester.

Validity checks

(*i*) Only user with role Data Entry Operator will be authorised to access the Students' Subject Choices Information Maintenance module.

(*ii*) The subject choice for Elective 1 and Elective 2 can be made only from the list of available choices for that semester.

Sequencing information

Students' Subject Choices Info for a particular student can be entered in the system only after Subject Info has been entered in the system for the given semester and the Student Info for that student has been entered in the system.

Students' Subject Choices Info for a particular student will have to be entered in the system before any marks info can be entered for that student in the given semester.

Error handling/response to abnormal situations

If any of the above validations/sequencing flow does not hold true appropriate error messages will be prompted to the user for doing the needful.

3.2.4 Marks Information Maintenance

Description

The system will maintain information about marks obtained by various students of different enrollment years in different semesters. The following information would be maintained:

Student Enrollment Number, Semester, Subject Code, Internal Marks, External Marks, Total Marks, and Credits.

The system will allow creation/modification/deletion of marks information and also have the ability to list all the available marks information for all students for a particular subject in the given semester.

Validity checks

(*i*) Only user with role Marks Entry Clerk will be authorised to access the Marks Information Maintenance module.

(*ii*) Internal Marks for any subject cannot be less than 0 and greater than 40.

(*iii*) External marks for any subject cannot be less than 0 and greater than 60.

(*iv*) Total marks in any subject will be calculated as: Internal Marks in that subject + External Marks in that subject.

(*v*) If the total Marks in a subject are > = 50, all the credit points associated with that subject will be given to the student, else the credit points earned by the student will be 0 for that subject.

Sequencing information

Marks Info for a particular student can be entered in the system only after Subject Info has been entered in the system for the given semester, the Student Info for that student has been entered in the system, and the Students' Subject Choice Info has been entered in the system for that student in the given semester.

Marks info for a particular student will have to be entered in the system before that student's mark-sheet can be generated.

Error handling/response to abnormal situations

If any of the above validations/sequencing flow does not hold true appropriate error messages will be prompted to the user for doing the needful.

3.2.5 Mark-sheet Generation

Description

The system will generate mark-sheet for every student in different semesters.

Mark-sheet will have the following format:

Name of the University
Name of Program
Semester <no.>
Mark-sheet

Student Enrollment No.____________ Student Name: ____________

S.No.	*Subject*	*Internal Marks (Out of 40)*	*External Marks (Out of 60)*	*Total Marks (Int. + Ext.)*	*Pass/Fail*	*Credits Earned*
1.						
2.						
3.						
4.						
5.						
6.						

Marks Grand Total: _____/600 Total Credits:_____
Result: (Pass/Fail)

Date: Signature of Controller of Examinations

There will be a 'Print' icon at the top of mark-sheet for printing the mark-sheet.

Validity checks

(i) Only user with role Marks Entry Clerk will be authorised to access the Mark-sheet Generation module.

Sequencing information

Marks-sheet for a particular student can be generated by the system only after Subject info has been entered in the system for the given semester, the Student Info for that student has been entered in the system, the Students' Subject Choice Info has been entered in the system for that student in the given semester, and the Marks Info has been entered for that student for the given semester.

Error handling/response to abnormal situations

If any of the above validations/sequencing flow does not hold true appropriate error messages will be prompted to the user for doing the needful.

3.2.6 Report Generation

Student list reports

For each year a report will be generated containing the list of students enrolled in that batch year.

Report format:

Name of University
Name of the Program
List of students entrolled in year xxxx

S.No.	*Student Enrollment Number*	*Student Name*
1.		
2.		

Students' subject choices list reports

For each batch year a report will be generated containing the list of students and their choices for Elective subjects in the selected semester. For Ist, IInd and IIIrd semesters the list will contain the names of elective subjects opted by each student. For IVth semester the list will contain the topic/subject area of dissertation for each student.

Report format (for Ist, IInd and IIIrd Semesters):

Name of University
Name of the Program

S.No.	*Student Enrollment Number*	*Student Name*	*Elective 1*		*Elective 2*	
			Code	*Name*	*Code*	*Name*
1.						
2.						

Report format (for IVth Semester):

Name of University
Name of the Program

S.No.	*Student Enrollment Number*	*Student Name*	*Topic/Subject Area of Dissertation*
1.			
2.			

Semester-wise mark lists

For each semester a mark list will be generated that will have the total marks (internal + external) of all students (enrolled in the selected Batch Year) of that semester in all subjects.

Report format:

Name of University

Name of the Program

S.No	*Enrollment Number*	*Subject 1 Total Marks*	*Subject 2 Total Marks*	*Subject 3 Total Marks*	*Subject 4 Total Marks*	*Subject 5 Total Marks*	*Subject 6 Total Marks*
1.							
2.							

Rank-wise list report

This report will be generated for each semester of every Batch Year. It will show the Grand Total marks, Percentage, Rank and Division secured of all students of that semester. The report will be sorted in increasing order of Percentage/Rank.

Report format:

Name of University

Name of the Program

S.No.	*Enroll. No.*	*Name*	*Grand Total Marks*	*%age*	*Rank*	*Division*
1.						
2.						

3.2.7 User Accounts Information Maintenance

Description

The system will maintain information about various users who will be able to access the system. The following information would be maintained:

User Name, User ID, Password, and Role.

Validity checks

(*i*) Only user with role Administrator will be authorised to access the User Accounts Information Maintenance module.

(*ii*) User Name cannot be blank.

(*iii*) User ID cannot be blank.

(*iv*) User ID should be unique for every user.

(*v*) Password cannot be blank.

(*vi*) Role cannot be blank.

Sequencing information

User Account for a particular user has to be created in order for the system to be accessible to that user. At system startup, only a default user account for 'Administrator' would be present in the system.

Error handling/response to abnormal situations

If any of the above validations/sequencing flow does not hold true, appropriate error messages will be prompted to the user for doing the needful.

3.3 Performance Requirements

None

3.4 Design Constraints

None

3.5 Software System Attributes

3.5.1 Security

The application will be password protected. Users will have to enter correct username, password and role in order to access the application.

3.5.2 Maintainability

The application will be designed in a maintainable manner. It will be easy to incorporate new requirements in the individual modules (*i.e.*, subject info, student info, students' subject choices info, marks info, report generation and user accounts info).

3.5.3 Portability

The application will be easily portable on any windows-based system that has MS-Access 2000 installed.

3.6 Logical Database Requirements

The following information will be placed in a database:

(*i*) **Subject info:** Subject Name, Code, Credit points, Type and Semester

(*ii*) **Student info:** Student enrollment number, Student Name, enrollment year

(*iii*) **Students' subject choice info:** Student Enrollment No, Semester, Choice of Elective 1 subject, Choice of Elective 2 subject

Marks info: Student Enrollment No., Semester, internal Marks in each subject, External Marks in each subject

User account info: User Name, User ID, Password, Role.

3.7 Other Requirements

None

REFERENCES

[BERR98] Berry D.M. & B. Lawrence, "*Requirements Engineering*", IEEE Software, Mar/April, 26–29, 1998.

[BITT03] Bittner K. and I spence, "*Use Case Modelling*", Pearson Education, 2003.

[BROO95] Brooks F.P., "No Silver Bullet", The Mythical Man Month: Essay on Software Engineering (Second Edition), Addison Wesley Longman, Reading, Mass, 179–209, 1995.

[CAIN75] Caine S. & E. Gordon, "*A Tool for Software Design*", Proceeding of the AFIPS, National Computer Conference, Vol. 47, AFIPS Press, Montvale, NJ, 271–276, 1975.

[CAPE94] Caper Jones, "*Charting the seas of Information Technology*", 1994.

[CARE90] Carey J. M., "*Prototyping: Alternative Systems Development Methodology*", Information and Software Technology, Vol. 32, No. 2, March 1990, 119–126.

[CERI88] Ceri S. et al., "*Software Prototyping by Relational Techniques: Expenses with Program Construction Systems*", IEEE Transaction on Software Engineering, Vol. 14, No. 11, Nov. 1988, 1597–1609.

[CHEN76] Chen P., "*The Entity–Relationship Model–Towards the Unified View of Data*", ACM Trans on database systems, March, 9–36, 1976.

[COAD89] Coad P. & E. Yourdon, "*OOA–Object Oriented Analysis*", Englewood Cliffs, N.J. Prentice Hall, 1989.

[COUN99] Councill B., "*Third–Party Testing and Stirrings of the New Software Engineering*", IEEE Software, Nov./Dec., 76–88, 1999.

[DAVI94] Davis A. & P. Hsia, "*Giving Voice to Requirements Engineering*", IEEE Software, March, 12–16, 1994.

[DAVI90] Davis A.M., "*Software Requirements Analysis and Specification*", PH, Englewood Cliffs, NJ, 1990.

[DEMA79] DeMacro T.,"*Structured Analysis and System Specification*", Englewood Cliffs, N.J., PH, 1979.

[HEKM87] HeKmatpour S., "*Experience Evolutionary Prototyping in Large Software Project*", ACM Software Engineering Notes, Vol. 12, No. 1, January, 1987, 38–41.

[HEME82] Hemenway K. & L. X. McCusker, "*Prototyping & Evaluating a User Interface*", Proc. of the 6th International Computer Software & Applications Conference, Chicago, Illinois, Nov. 1982, 175–180.

[HICK03] Hickey A.M. and A.M. Davis, "*Requirements Elicitation and Elicitation Technique Selection: A Model for Two Knowledge Intensive Software Development Processes*",Proc. of 36th Hawai Int. conf. on system science, 2003.

[HOFF99] Hoffer J.A. et al., "*Modern Systems Analysis and Design*", Addison–Wesley, 1999.

[HOOP89] Hooper W.J., "*Languages Features for Prototyping and Simulation Support of the Software Life Cycle*", Computer Languages, Vol. 14, No. 2, Feb. 1989, 83–92.

[HSIA93] Hsia P., A. Davis & D. Kung, "*Status Report: Requirements Engineering*", IEEE Software, 75–79, Nov. 1993.

[IBMG2K] IBM Global Services India Pvt. Ltd., Golden Tower, Airport Road, Bangalore, Letter of Bindu Subramani Segment Manager Corporate Training, March 9, 2000.

[IEEE87] IEEE Standard for Project Management Plans (ANSI), IEEE Std., 1058.1, 1987.

[IEEE94] IEEE Standards for Requirements Engineering, 1994.

[IEEE97] IEEE Standard Taxonomy for Software Engineering Standards (ANSI), 1997.

[IEEE93] IEEE guide to software requirements specifications (std. 830–1993). QS/IRM/Private Initial/QA/QA Plan/Sig Plan, DOC, 1993.

[MACA96] Macaulay L., "*Requirements Engineering*", Springer, 1996.

[MANT88] Mantei M. & Teorey T., "*Cost / Benefit for Incorporating Human Factors in the Software Life Cycle*", Communications of the ACM 31, 4, April, 428–439, 1988.

[MARC88] Marca D. & McGowan C., "*Structured Analysis and Design Techniques*", New York, MH., 1988.

[MARC88] Marco D. & McGowan C., "*Structured Analysis & Design Techniques*", NY, McGraw Hill, 1988.

[MARC88] Marco D. & McGowan, "*Structured Analysis & Design Techniques*", NY, McGraw Hill, 1988.

[MUSA87] Musa J. et al., "*Software Reliability*", New York, McGraw Hill, 1987.

[PRES2K] Pressman R.S., "*Software Engineering*", McGraw Hill, 2000.

[RATC88] Ratcliff B., "*Early and Not So Early Prototyping–Rationale and Tool Support*", Proc. of the 12th Annual Int. Computer Software & Application Conference, Chicago, Illinois, IEEE Computer Society Press, Oct, 1988, 127–134.

[ROBE02] Robert T. Futrell et. al. "*Quality Software Project Management*", Pearson Education Asia, 2002.

[ROSS77] Ross D., "*Structured Analysis: A Language for Communicating Ideas*", IEEE Trans on Software Engineering, 3,1, Jan, 6–15, 1977.

[SAGE90] Sage A.P. & Palma J.D., "*Software System Engineering*", John Wiley & Sons, 1990.

[SAWY99] Sawyer P. et al., "*Capturing the benefits of Requirement Engineering*", IEEE Software, 78–85, March/April, 1999.

[SKEL86] Skelton S., "*Measurements of Migratability and Transportability*", ACM Software Engineering Notes, 11, 1, 29–34, 1986.

[SOMM96] Sommerville I., "*Software Engineering*", Addison Wesley, 1996.

[SOMM01] Sommerville I., "*Software Engineering*", Pearson Education, 2001.

[STEV98] Steve McConnell, "*Feasibility studies*", IEEE software, May/June, 119-120, 1998.

[SOMM05] Sommerville I, "*Software Engineering* (7th ed.)", Pearson Education, 2005.

[THAY97] Thayer R.H. & Dorfman M., "*Software Requirements Engineering*", IEEE Computer Society, LA, 1997.

[TOZE87] Tozer J. E., "*Prototyping as a System Development Methodology: Opportunities and Pitfalls*", Information and Software Technology, Vol. 29, No. 5, June 1987, 265–269.

[WIEG99] Wiegers K.E., "*Software Requirements*", Microsoft Press, Washington, USA, 1999.

[YOUR79] Yourdon E. & Constantine L., "*Structured Design*", Englewood Cliffs, NJ, Prentice-Hall, 1979.

[ZULT92] Zultner R., "*Quality Function Deployment for Software: Satisfying Customers*" American Programmer, February 28–41' 1992.

MULTIPLE CHOICE QUESTIONS

Note. *Choose the most appropriate answer of the following questions.*

3.1. Which one is not a step of requirement engineering ?
(*a*) requirements elicitation (*b*) requirements analysis
(*c*) requirements design (*d*) requirements documentation

3.2. Requirements elicitation means
(*a*) gathering of requirements (*b*) capturing of requirements
(*c*) understanding of requirements (*d*) all of the above.

3.3. SRS stands for
(*a*) software requirements specification (*b*) system requirements specification
(*c*) systematic requirements specifications (*d*) none of the above.

3.4. SRS document is for
(*a*) "what" of a system ? (*b*) how to design the system ?
(*c*) costing and scheduling of a system (*d*) system's requirement.

3.5. Requirements review process is carried out to
(*a*) spend time in requirements gathering (*b*) improve the quality of SRS
(*c*) document the requirements (*d*) none of the above.

3.6. Which one of the statements is not correct during requirements engineering ?
(*a*) requirements are difficult to uncover
(*b*) requirements are subject to change
(*c*) requirements should be consistent
(*d*) requirements are always precisely known.

3.7. Which one is not a type of requirements ?
(*a*) known requirements (*b*) unknown requirements
(*c*) undreamt requirements (*d*) complex requirements.

3.8. Which one is not a requirements elicitation technique
(*a*) interviews (*b*) the use case approach
(*c*) fAST (*d*) data flow diagram

3.9. FAST stands for
(*a*) functional Application Specification Technique
(*b*) fast Application Specification Technique
(*c*) facilitated Application Specification Technique
(*d*) none of the above.

3.10. QFD in requirement engineering stands for
(*a*) quality function design (*b*) quality factor design
(*c*) quality function development (*d*) quality function deployment.

3.11. Which is not a type of requirements under quality function deployment

(*a*) normal requirements (*b*) abnormal requirements

(*c*) expected requirements (*d*) exciting requirements.

3.12. Use case approach was developed by

(*a*) I. Jacobson and others (*b*) J.D. Musa and others

(*c*) B. Littlewood (*d*) none of the above.

13. Context diagram explains

(*a*) the overview of the system (*b*) the internal view of the system

(*c*) the entities of the system (*d*) none of the above.

3.14. DFD stands for

(*a*) data flow design (*b*) descriptive functional design

(*c*) data flow diagram (*d*) none of the above.

3.15. Level-O DFD is similar to

(*a*) use case diagram (*b*) context diagram

(*c*) system diagram (*d*) none of the above.

3.16. ERD stands for

(*a*) entity relationship diagram (*b*) exit related diagram

(*c*) entity relationship design (*d*) exit related design.

3.17. Which is not a characteristic of a good SRS ?

(*a*) correct (*b*) complete

(*c*) consistent (*d*) brief.

3.18. Outcome of requirements specification phase is

(*a*) design document (*b*) software requirements specification

(*c*) test document (*d*) none of the above.

3.19. The basic concepts of ER model are:

(*a*) entity and relationship (*b*) relationships and keys

(*c*) entity, effects and relationship (*d*) entity, relationship and attribute.

3.20. The DFD depicts

(*a*) flow of data (*b*) flow of control

(*c*) both (*a*) and (*b*) (*d*) none of the above.

3.21. Product features are related to :

(*a*) functional requirements (*b*) non-functional requirements

(*c*) interface requirements (*d*) none of the above.

3.22. Which one is a quality attribute ?

(*a*) reliability (*b*) availability

(*c*) security (*d*) all of the above.

3.23. IEEE Std. for SRS is :

(*a*) 837-1998 (*b*) 830-1998

(*c*) 832-1998 (*d*) 839-1998.

3.24. Which is not a non-functional requirement ?

(*a*) efficiency (*b*) reliability

(*c*) product features (*d*) stability.

3.25. APIs stands for

(a) application performance Interfaces • (b) application programming Interfaces

(c) application programming integration (d) application performance integration.

EXERCISE

3.1. Discuss the significance and use of requirement engineering. What are the problems in the formulation of requirements ?

3.2. Requirements analysis is unquestionably the most communication intensive step in the software engineering process. Why does the communication path frequently break down ?

3.3. What are crucial process steps of requirement engineering ? Discuss with the help of a diagram.

3.4. Discuss the present state of practices in requirement engineering. Suggest few steps to improve the present state of practice.

3.5. Explain the importance of requirements. How many types of requirements are possible and why ?

3.6. Describe the various steps of requirements engineering. Is it essential to follow these steps ?

3.7. What do you understand with the term "requirements elicitation" ? Discuss any two techniques in detail.

3.8. List out requirements elicitation techniques. Which one is most popular and why ?

3.9. Describe facilitated application specification technique (FAST) and compare this with brainstorming sessions.

3.10. Discuss quality function deployment technique of requirements elicitation. Why an importance or value factor is associated with every requirement ?

3.11. Explain the use case approach of requirements elicitation. What are use-case guidelines ?

3.12. What are components of a use case diagram. Explain their usage with the help of an example.

3.13. Consider the problem of library management system and design the following:

(i) Problem statement

(ii) Use case diagram

(iii) Use cases.

3.14. Consider the problem of railway reservation system and design the following:

(i) Problem statement

(ii) Use case diagram

(iii) Use cases.

3.15. Explain why a many to many relationship is to be modeled as an associative entity ?

3.16. What are the linkages between data flow and E–R diagrams ?

3.17. What is the degree of a relationship ? Give an example of each of the relationship degree.

3.18. Explain the relationship between minimum cardinality and optional and mandatory participation.

3.19. An airline reservation is an association between a passenger, a flight, and a seat. Select a few pertinent attributes for each of these entity types and represent a reservation in an E–R diagram.

3.20. A department of computer science has usual resources and usual users for these resources. A software is to be developed so that resources are assigned without conflict. Draw a DFD specifying the above system.

3.21. Draw a DFD for result preparation automation system of B. Tech. courses (or MCA program) of any university. Clearly describe the working of the system. Also mention all assumptions made by you.

3.22. Write short notes on

(*i*) Data flow diagram

(*ii*) Data dictionary.

3.23. Draw a DFD for borrowing a book in a library which is explained below: "A borrower can borrow a book if it is available else he/she can reserve for the book if he/she so wishes. He/she can borrow a maximum of three books".

3.24. Draw the E–R diagram for a hotel reception desk management.

3.25. Explain why, for large software systems development, is it recommended that prototypes should be "throw-away" prototype ?

3.26. Discuss the significance of using prototyping for reusable components and explain the problems, which may arise in this situation.

3.27. Suppose a user is satisfied with the performance of a prototype. If he/she is interested to buy this for actual work, what should be the response of a developer ?

3.28. Comment on the statement: "The term throw-away prototype is in appropriate in that these prototypes expand and enhance the knowledge base that is retained and incorporated in the final prototype; therefore they are not disposed of or thrown away at all."

3.29. Which of the following statements are ambiguous ? Explain why.

(*a*) The system shall exhibit good response time.

(*b*) The system shall be menu driven.

(*c*) There shall exist twenty-five buttons on the control panel, numbered PF1 to PF25.

(*d*) The software size shall not exceed 128K of RAM.

3.30. Are there other characteristics of an SRS (besides listed in section 3.4.2) that are desirable ? List a few and describe why ?

3.31. What is software requirements specification (SRS) ? List out the advantages of SRS standards. Why is SRS known as the black box specification of a system ?

3.32. State the model of a data dictionary and its contents. What are its advantages ?

3.33. List five desirable characteristics of a good SRS document. Discuss the relative advantages of formal requirement specifications. List the important issues, which an SRS must address.

3.34. Construct an example of an inconsistent (incomplete) SRS.

3.35. Discuss the organisation of a SRS. List out some important issues of this organisation.

3.36. Discuss the difference between the following :

(*a*) Functional and Nonfunctional Requirements.

(*b*) User and system requirements.

Software Project Planning 4

After the finalisation of SRS, we would like to estimate size, cost and development time of the project. Also, in many cases, customer may like to know the cost and development time even prior to finalisation of the SRS. Hence, whether we estimate before SRS or after SRS, it would always be very critical and crucial decision for the project. Estimation of cost and development time are the key issues during project planning. The correlation between cost, development time, and the planning process can best be illustrated by an example.

Suppose we want to put an addition on our home. After deciding what we want and getting several quotations, most of which are around 2.5 lacs, we pick up a builder who offers to do the job in two months for 2.0 lacs. We sign an agreement and the builder starts the work. After about a month into the job, the builder comes and explains that because of the problems the job will take an extra month and cost an additional 0.5 lacs. This creates several problems. First, we badly need space and another month of delay is a real inconvenience. Second, we have already arranged for a loan and do not know from where we can get this additional amount of Rs. 0.5 lac. Third, if we get a lawyer and decide to fight the builder in court, all work on the job will stop for many months while the case is decided. Fourth, it would take a great deal of time and probably cost even more to switch to a new builder in the middle of the job.

On exploration, we conclude that the real problem is that the builder did a sloppy job of planning. Builder might have forgot to include some major costs like the labour or materials to do the woodwork or final plastering and painting. Because other quotations were close to Rs. 2.5 lacs, we know that this is pretty fair price. At this point, we have no option but to try to negotiate a lower price but will continue with the current builder. However, we would neither use this builder again, nor would probably recommend the builder to anyone else [HUMP95].

This is the essential issue in the planning process; being able to make plans that accurately represent what we can do. Business operates on commitments, and commitments require plans. The failure of many large software projects in the 1960s and early 1970s highlighted this problem of poor planning. The delivered software was late, unreliable, costed several times the original estimates and often exhibited poor performance characteristics. These projects did not fail because managers or developers were incompetent. The fault was in the approach of planning that was used. Planning techniques derived from small-scale projects did not scale up to large systems development.

Software managers are responsible for planning and scheduling project development. They supervise the work to ensure that it is carried out to the required standards. They monitor progress to check that the development is on time and within budget. Good managers cannot guarantee project success. However, bad managers usually result in project failure. Usually, the software is delivered late, costs more than originally estimated and fails to meets its re-

quirements [SOMM95]. The project planning must incorporate the major issues like size and cost estimation, scheduling, project monitoring and reviews, personnel selection and evaluation, and risk management.

In order to conduct a successful software project, we must understand [PRES2K]

- scope of work to be done
- the risk to be incurred
- the resources required
- the task to be accomplished
- the cost to be expended
- the schedule to be followed

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired. The various steps of planning activities are illustrated in Fig. 4.1. As shown, first activity is to estimate the size of the project. The size is the key parameter for the estimation of other activities. It is an input to all costing models for the estimation of cost, development time and schedule for the project. If size estimation is not reasonable, it may have serious impact on the other estimation activities.

Resources requirements are estimated on the basis of cost and development time. Project scheduling may prove to be very useful for controlling and monitoring the progress of the project. This is dependent on the resources and development time.

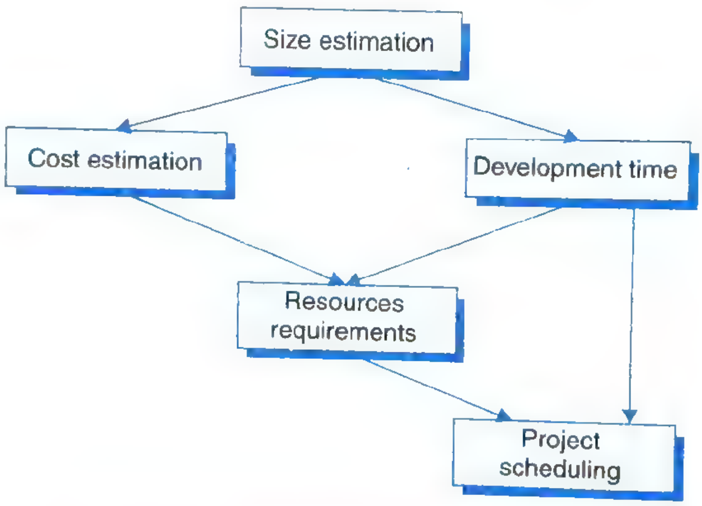


Fig. 4.1: Activities during software project planning

4.1 SIZE ESTIMATION

Some programs are written in C, some in PASCAL, others in FORTRAN, and still others in assembly language. Some programs are for GUI applications and some are for batch processing. Some are written using the latest software engineering techniques, while others are developed without adequate planning. Some programs are well documented with carefully crafted internal comments, other programs are written in a quick and dirty fashion with no comments at all. But, there is one characteristic that all programs share—they all have size [CONT86].

The estimation of size is very critical and difficult area of the project planning. It has been recognised as a crucial step from the very beginning. The difficulties in establishing units for measuring size lie in the fact that the software is essentially abstract: it is difficult to identify the size of a system. Other engineering disciplines have the advantage that a bridge or a building or a road can be seen and touched, they are (sometimes literally) concrete. Many attempts have been made at establishing a unit of measure for size. The more widely known are given below.

4.1.1 Lines of Code (LOC)

This was the first measurement attempted. It has the advantage of being easily recognisable, seen and therefore counted. Although this may seem to be a simple metric that can be counted algorithmically, there is no general agreement about what constitutes a line of code. Early users of lines of code did not include data declarations, comments, or any other lines that did not result in object code. Later users decided to include declarations and other unexecutable statements but still excluded comments and blank lines. The reason for this shift is the recognition that contemporary code can have 50% or more data statements and that bugs occur as often in such statements as in real code. For example, in the function shown in Fig. 4.2, if LOC is simply a count of the number of lines then Fig. 4.2 contains 18 LOC [CONT86].

1.	int. sort (int x[ ], int n)
2.	{
3.	int i, j, save, im1;
4.	/* This function sorts array × in ascending order */
5.	If (n < 2) return 1;
6.	for (i = 2; i < = n; i++)
7.	{
8.	im1 = i – 1;
9.	for (j = 1; j < = im; j++)
10.	if (x[i] < x[j])
11.	{
12.	Save = x[i];
13.	x[i] = x[j];
14.	x[j] = save;
15.	}
16.	}
17.	return 0;
18.	}

Fig. 4.2: A function for sorting an array in ascending order

But most researchers agree that the LOC metric should not include comments or blank lines. Since these are really internal documentation and their presence or absence does not affect the functions of the program. Moreover, comments and blank lines are not as difficult to

construct as program lines. The inclusion of comments and blank lines in the count may encourage developers to introduce artificially many such lines in project development in order to create the illusion of high productivity, which is normally measured in LOC/PM (lines of code person-month). When comments and blank lines are ignored, the program in Figs. 4.2 contains 17 LOC.

However, there is a fundamental reason for including comments in the program. The quality of comments materially affects maintenance costs because maintenance person will depend on the comments more than anything else to do the job. Conversely, too many blank lines and comments with poor readability and understandability will increase maintenance effort. The problem with including comments is that we must be able to distinguish between useful and useless comments, and there is no rigorous way to do that. Therefore, it is always advisable not to consider comments and blank lines while counting for LOC.

Furthermore, if the main interest is the size of the program for specific functionality, it may be reasonable to include executable statements. The only executable statements in Fig 4.2 are in lines 5–17 leading to a count of 13. The differences in the counts are 18 to 17 to 13. One can easily see the potential for major discrepancies for large programs with many comments or programs written in languages that allow a large number of descriptive but non-executable statements. Conte [CONT86] has defined lines of code as:

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declarations, and executable and non-executable statements".

This is the predominant definition for lines of code used by researchers. By this definition Fig. 4.2 has 18 LOC.

There are some disadvantages of this simple method of counting. LOC is language dependent. A line of assembler is not the same as a line of COBOL. They also reflect what the system is rather than what it does. Measuring systems by the number of lines of code is rather like measuring a building by the number of bricks involved in construction; useful when deciding on the type and number of brick layers but useless in describing the building as a whole. Buildings are normally described in terms of facilities, the number and size of rooms, and their total areas in square feet or meters.

While lines of code have their uses (*e.g.*, in estimating programming time for a program during the build phase of a project), their usefulness is limited for other tasks like functionality, complexity, efficiency, etc. If counting LOC is similar to counting bricks in a building, then what is needed is some way of expressing the system in a way that is analogous to counting the number and size of rooms and their total area in square feet or meters.

4.1.2 Function Count

Measuring software size in terms of lines of code is analogous to measuring a car stereo by the number of resistors, capacitors and integrated circuits involved in its production. The number of components is useful in predicting the number of assembly line staff needed, but it does not say anything about the functions available in the finished stereo. When dealing with customers, the manufacturer talks in terms of functions available (*e.g.*, digital tuning) and not in terms of components (*e.g.*, integrated circuits).

Alan Albrecht while working for IBM, recognised the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem [ALBR79, ALBR83]. It measures functionality from the users point of view, that is, on the basis of what the user requests and receives in return. Therefore, it deals with the functionality being delivered, and not with the lines of code, source modules, files, etc. Measuring size in this way has the advantage that size measure is independent of the technology used to deliver the functions. In other words, two identical counting systems, one written in 4 GL and the other in assembler, would have the same function count. This makes sense to the user, because the object is to buy an accounting system, not lines of assembler and it makes sense to the IT department, because they can measure the performance differences between the assembler and 4GL environments [STEP95].

Function point measures functionality from the users point of view, that is, on the basis of what the user requests and receives in return from the system. The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units

- Inputs : information entering the system.
- Outputs : information leaving the system.
- Enquiries : requests for instant access to information.
- Internal logical files : information held within the system.
- External interface files : Information held by other systems that is used by the system being analyzed.

The FPA functional untis are shown in Fig. 4.3.

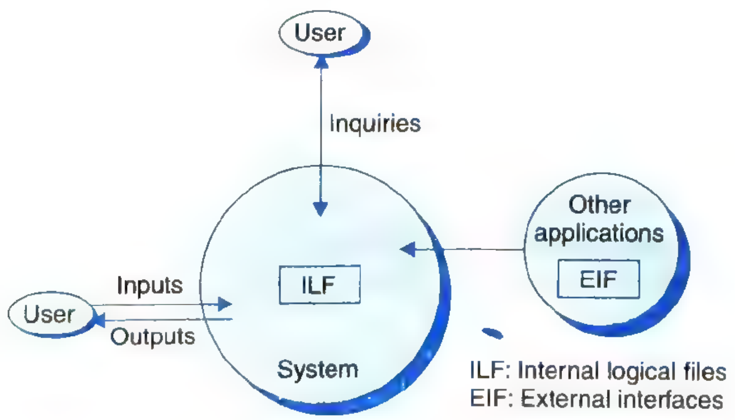


Fig. 4.3: FPAs functional units

The five functional units are divided in two categories:

(i) **Data function types**

- *Internal Logical files (ILF):* A user identifiable group of logically related data or control information maintained within the system.
- *External Interface files (EIF):* A user identifiable group of logically related data or control information referenced by the system, but maintained within another system. This means that EIF counted for one system, may be an ILF in another system.

(*ii*) **Transactional function types**

- *External Input (EI):* An EI processes data or control information that comes from outside the system. The EI is an elementary process, which is the smallest unit of activity that is meaningful to the end user in the business.
- *External Output (EO):* An EO is an elementary process that generates data or control information to be sent outside the system.
- *External Inquiry (EQ):* An EQ is an elementary process that is made up of an input-output combination that results in data retrieval.

Special features

- Function point approach is independent of the language, tools, or methodologies used for implementation; *i.e.*, they do not take into consideration programming languages, data base management systems, processing hardware or any other data base technology.
- Function points can be estimated from requirement specification or design specifications, thus making it possible to estimate development effort in early phases of development.
- Function points are directly linked to the statement of requirements; any change of requirements can easily be followed by a re-estimate [INCE89].
- Function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

This method resolves many of the inconsistencies that arise when using lines of code as a software size measure [MATS94].

Counting function points

The five functional units are ranked according to their complexity *i.e.*, Low, Average, or High, using a set of prescriptive standards. Organisations that use FP methods develop criteria for determining whether a particular entry is Low, Average or High. Nontheless, the determination of complexity is somewhat subjective.

After classifying each of the five function types, the Unadjusted Function Points (UFP) are calculated using predefined weights for each function type as given in Table 4.1.

Table 4.1: Functional units with weighting factors

Functional units	*Weighting factors*		
	Low	*Average*	*High*
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
Internal logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 4.2: UFP calculation table

Functional units	*Count complexity*			*Complexity totals*	*Functional unit totals*
External Inputs (EIs)	☐	Low × 3	=	☐	
	☐	Average × 4	=	☐	
	☐	High × 6	=	☐	☐
External Outputs (EOs)	☐	Low × 4	=	☐	
	☐	Averge × 5	=	☐	
	☐	High × 7	=	☐	☐
External Inquiries (EQs)	☐	Low × 3	=	☐	
	☐	Average × 4	=	☐	
	☐	High × 6	=	☐	☐
Internal logical files (ILFs)	☐	Low × 7	=	☐	
	☐	Average × 10	=	☐	
	☐	High × 15	=	☐	☐
External Interface files (EIFs)	☐	Low × 5	=	☐	
	☐	Average × 7	=	☐	
	☐	High × 10	=	☐	☐
Total Unadjusted Function Point Count					☐

The weighting factors are identified (as per Table 4.1) for all functional units and multiplied with the functional units accordingly. The procedure for the calculation of Unadjusted Function Point (UFP) is given in Table 4.2.

The procedure for the calculation of UFP in mathematical form is given below:

$$\text{UFP} = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

where i indicates the row and j indicates the column of Table 4.1.

w_{ij} : It is the entry of the i^{th} row and j^{th} column of the Table 4.1.

Z_{ij} : It is the count of the number of functional units of Type i that have been classified as having the complexity corresponding to column j.

Organisations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of complexity is somewhat subjective.

The final number of function points is arrived at by multiplying the UFP by an adjustment factor that is determined by considering 14 aspects of processing complexity which are given in Table 4.3. This adjustment factor allows the UFP count to be modified by at most ± 35%. The final adjusted FP count is obtained by using the following relationship.

FP = UFP * CAF

Where CAF is complexity adjustment factor and is equal to $[0.65 + 0.01 \times \Sigma F_i]$. The F_i (i = 1 to 14) are the degrees of influence and are based on responses to questions noted in Table 4.3.

Table 4.3: Computing function points

Rate each factor on a scale of 0 to 5.

0	1	2	3	4	5
No Influence	*Incidental*	*Moderate*	*Average*	*Significant*	*Essential*

Number of factors considered (F_i)

1. Does the system require reliable backup and recovery?
2. Is data communication required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing heavily utilized operational environment?
6. Does the system require on line data entry?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on line?
9. Is the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organisations?
14. Is the application designed to facilitate change and ease of use by the user?

Uses of function points

The collection of function point data has two primary motivations. One is the desire by managers to monitor levels of productivity, for example, number of function points achieved per work hour expended. From this perspective, the manager is not concerned with when the function point counts are made, but only that the function points accurately describe the size of the final software project. In this instance, function points have an advantage over LOC in that they provide a more objective measure of software size by which to assess productivity.

Another use of function points is in the estimation of software development cost. There are only a few studies that address this issue, though it is arguably the most important potential use of function point data.

Functions points may compute the following important metrics:

Productivity	=	FP/persons-months
Quality	=	Defects/FP
Cost	=	Rupees /FP
Documentation	=	Pages of documentation per FP

These metrics are controversial and are not universally acceptable. There are standards issued by the International Function Point User Group (IFPUG, covering the Albrecht method) and the United Kingdom Function Point User Group (UFPGU, covering the MK11 method). An ISO standard for function point methods is also being developed.

The function point method continues to be refined. So if we intend to use it we should obtain copies of the latest international function point user group (IFPUG) guidelines and standards. IFPUG is the fastest growing non-profit software metrics user group in the world with an annul growth rate of 30%. Today it has grown to over 800 corporate members and over 1500 individual members in more than 50 countries.

Example 4.1

Consider a project with the following functional units:

Number of user inputs = 50
Number of user outputs = 40
Number of user enquiries = 35
Number of user files = 06
Number of external interfaces = 04

Assume all complexity adjustment factors and weighting factors are average.

Compute the function points for the project.

Solution

We know

$$\text{UFP} = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij}\, w_{ij}$$

$$\begin{aligned}
\text{UFP} &= 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7 \\
&= 200 + 200 + 140 + 60 + 28 = 628 \\
\text{CAF} &= (0.65 + 0.01\ \Sigma F_i) \\
&= (0.65 + 0.01(14 \times 3)) = 0.65 + 0.42 = 1.07 \\
\text{FP} &= \text{UFP} \times \text{CAF} \\
&= 628 \times 1.07 = 672.
\end{aligned}$$

Example 4.2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average external inequiries, and a value of complexity adjustment factor of 1.10.

What are the unadjusted and adjusted function point counts ?

Solution

Unadjusted function point counts may be calculated using as:

$$\text{UFP} = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

$$= 10 \times 3 + 12 \times 7 + 20 \times 7 + 15 + 10 + 12 \times 4$$

$$= 30 + 84 + 140 + 150 + 48$$

$$= 452$$

$$\text{FP} = \text{UFP} \times \text{CAF}$$

$$= 452 \times 1.10 = 497.2.$$

Example 4.3

Consider a project with the following parameters.

(*i*) External Inputs:

(*a*) 10 with low complexity

(*b*) 15 with average complexity

(*c*) 17 with high complexity.

(*ii*) External Outputs:

(*a*) 6 with low complexity

(*b*) 13 with high complexity.

(*iii*) External Inquiries:

(*a*) 3 with low complexity

(*b*) 4 with average complexity

(*c*) 2 with high complexity.

(*iv*) Internal logical files:

(*a*) 2 with average complexity

(*b*) 1 with high complexity.

(*v*) External Interface files:

(*a*) 9 with low complexity.

In addition to above, system requires

(*i*) Significant data communication

(*ii*) Performance is very critical

(*iii*) Designed code may be moderately reusable

(*iv*) System is not designed for multiple installations in different organisations.

Other complexity adjustment factors are treated as average. Compute the function points for the project.

Solution

Unadjusted function points may be counted using Table 4.2.

Functional units	*Count*	*Complexity*				*Complexity totals*	*Functional unit totals*
External	10	Low	×	3	=	30	
Inputs	15	Average	×	4	=	60	
(EIs)	17	High	×	6	=	102	192
External	6	Low	×	4	=	24	
Outputs	0	Average	×	5	=	0	
(EOs)	13	High	×	7	=	91	115
External	3	Low	×	3	=	9	
Inquiries	4	Average	×	4	=	16	
(EIs)	2	High	×	6	=	12	37
Internal	0	Low	×	7	=	0	
Logical	2	Average	×	10	=	20	
Files (ILFs)	1	High	×	15	=	15	35
External	9	Low	×	5	=	45	
Interface	0	Avergae	×	7	=	0	
Files (EIFs)	0	High	×	10	=	0	45
		Total unadjusted function point count =					424

The factors given in Table 4.3 may be calculated as:

$$\sum_{i=1}^{14} F_i = 3 + 4 + 3 + 5 + 3 + 3 + 3 + 3 + 3 + 3 + 2 + 3 + 0 + 3 = 41$$

4 + 5 + 2 + 0
+ 3 - -

$$\text{CAF} = (0.65 + 0.01 \times \Sigma F_i)$$
$$= (0.65 \times 0.01 \times 41)$$
$$= 1.06$$
$$\text{FP} = \text{UFP} \times \text{CAF}$$
$$= 424 \times 1.06$$
$$= 449.44$$

Hence $\boxed{\text{FP} = 449}$

4.2 COST ESTIMATION

For any new software project, it is necessary to know how much will it cost to develop and how much development time will it take. These estimates are needed before development is initiated. But how is this done? In many cases estimates are made using past experience as the only guide. However, in most of the cases projects are different and hence past experience alone may not be enough. A number of estimation techniques have been developed and are having following attributes in common.

- Project scope must be established in advance
- Software metrics are used as a basis from which estimates are made
- The project is broken into small pieces which are estimated individually

To achieve reliable cost and schedule estimates, a number of options arise:

- Delay estimation until late in project (obviously we can achieve 100% accurate estimates after project is complete!)
- Use simple decomposition techniques to generate project cost and schedule estimates
- Develop empirical models for estimation
- Acquire one or more automated estimation tools

Unfortunately, the first option, however, attractive, is not practical. Cost estimates must be provided up front. However, we should recognise that the longer we wait, more we know, and more we know, the less likely, are we to make serious errors in our estimates [PRES2K]

4.3 MODELS

The model is concerned with the representation of the process to be estimated. A model may be static or dynamic. In a static model, a unique variable (say, size) is taken as a key element for calculating all others (say, cost, time). The form of equation used is the same for all calculations. In a dynamic model, all variables are interdependent and there is no basic variable as in the static model.

When a model makes use of a single basic variable to calculate all others it is said to be a *single-variable* model. In some models, several variables are needed to describe the software development process, and selected equations combine these variables to give the estimate of time and cost. These models are called *multivariable*. The variables, single or multiple, that are input to the model to predict the behaviour of a software development are called *predictors*. The choice and handling of these predictors are most crucial activity in estimating methodology [LOND87].

4.3.1 Static, Single Variable Models

Methods using this model use an equation to estimate the desired values such as cost, time, effort, etc. They all depend on the same variable used as predictor (say, size). An example of the most common equation is

$$C = a\,L^b \tag{4.1}$$

where C is the cost (effort expressed in any unit of manpower, for example, person-months) and L is the size generally given in the number of lines of code. The constants, a & b are

derived from the historical data of the organisation. Since a and b depend on the local development environment, these models are not transportable to different organisations.

The Software Engineering Laboratory of the University of Maryland has established a model, the SEL model, for estimating its own software productions. This model [BASL80] is a typical example of a static single-variable model.

$$E = 1.4\ L^{0.93} \tag{4.2}$$

$$DOC = 30.4\ L^{0.90} \tag{4.3}$$

$$D = 4.6\ L^{0.26} \tag{4.4}$$

Effort (E in Person-months), documentation (DOC, in number of pages) and duration D, in months) are calculated from the number of lines of code (L, in thousands of lines) used as a predictor.

4.3.2 Static, Multivariable Models

Although these models are often based on equation (4.1), they actually depend on several variables representing various aspects of the software development environment, for example, methods used, user participation, customer oriented changes, memory constraints, etc. The model developed by Walston and Felix at IBM [WALS77] provides a relationship between delivered lines of source code (L in thousands of lines) and effort E (E in person- months) and is given by the following equation:

$$E = 5.2\ L^{0.91} \tag{4.5}$$

In the same fashion, the duration of the development (D in months) in given by

$$D = 4.1\ L^{0.36} \tag{4.6}$$

Data collected on 60 software projects, representing a wide variety of applications and size (ranging from 4000 to 467000 lines of code), shows a relationship between productivity (expressed in number of lines of source code per person months) and a productivity index I.

The productivity index uses 29 variables which are found to be highly correlated to productivity as follows:

$$I = \sum_{i=1}^{29} W_i X_i \tag{4.7}$$

where W_i is a factor weight for the i^{th} variable and $X_i = \{-1, 0, +1\}$. The estimator gives X_i one of the values – 1, 0 or +1 depending on whether the variable decreases, has no effect, or increases the productivity respectively. The terms of equation (4.7) are then added up to give the productivity index. A productivity range can be obtained for the project by using a productivity versus index chart [LOND87].

Example 4.4

Compare the Walston-Felix model [equation (4.5) and equation (4.6)] with the SEL model [equation (4.2) and equation (4.4)] on a software development expected to involve 8 person-years of effort [LOND87].

(*a*) Calculate the number of lines of source code that can be produced.

(*b*) Calculate the duration of the development.

(c) Calculate the productivity in LOC/PY.

(d) Calculate the average manning.

Solution

The amount of manpower involved = 8 PY = 96 person-months

(a) Number of lines of source code can be obtained by reversing equation (4.2 and equation (4.5) to give:

$$L = (E/a)^{1/b}$$

Then

$$L(SEL) = (96/1.4)^{1/0.93} = 94264 \text{ LOC}$$
$$L(W\text{-}F) = (96/5.2)^{1/0.91} = 24632 \text{ LOC}$$

(b) Duration in months can be calculated by means of equation (4.4) and equation (4.6)

$$D(SEL) = 4.6^{0.26}\, L$$
$$= 4.6(94.264)^{0.26} = 15 \text{ months}$$
$$D(W\text{-}F) = 4.1\, L^{0.36}$$
$$= 4.1\,(24.632)^{0.36} = 13 \text{ months}$$

(c) Productivity is the lines of code produced per person/month (year).

$$P(SEL) = \frac{94264}{8} = 11783 \text{ LOC/Person-Years}$$
$$P(W\text{-}F) = \frac{24632}{8} = 3079 \text{ LOC/Person-Years}$$

(d) Average manning is the average number of persons required per month in the project.

$$M(SEL) = \frac{96 \text{ P-M}}{15 \text{ M}} = 6.4 \text{ Persons}$$
$$M(W\text{-}F) = \frac{96 \text{ P-M}}{13 \text{ M}} = 7.4 \text{ Persons}$$

If we look at the value of "L", it seems that SEL can produce four times as much software as IBM for the same manpower and time scale.

4.4 THE CONSTRUCTIVE COST MODEL (COCOMO)

This model gained rapid popularity following the publication of B.W. Boehm's excellent book *Software Engineering Economics* in 1981 [BOEH81]. COCOMO is a hierarchy of software cost estimation models, which include basic, intermediate and detailed sub models.

4.4.1 Basic Model

The basic model aims at estimating, in a quick and rough fashion, most of the small to medium sized software projects. Three modes of software development are considered in this model: organic, semi-detached ~~and embedded~~.

In the organic mode, a small team of experienced developers develops software in a very familiar environment. The size of the software development in this mode ranges from small

(a few KLOC) to medium (a few tens of KLOC), while in other two modes the size ranges from small to very large (a few hundreds of KLOC).

In the embedded mode of software development, the project has tight constraints, which might be related to the target processor and its interface with the associated hardware. The problem to be solved is unique and so it is often hard to find experienced persons, as the same does not usually exist.

The *semi detached* mode is an intermediate mode between the organic mode and embedded mode. The comparision of all three modes is given in Table 4.4.

Table 4.4: The comparison of three COCOMO modes

Mode	*Project size*	*Nature of project*	*Innovation*	*Deadline of the project*	*Development environment*
Organic	Typically 2 – 50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar projects. For Example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For Example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

Depending on the problem at hand, the team might include a mixture of experienced and less experienced people with only a recent history of working together. The basic COCOMO equations take the form

$$E = a_b(\text{KLOC})^{b_b} \tag{4.8}$$

$$D = c_b(E)^{d_b} \tag{4.9}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in Table 4.4(a).

Table 4.4(a): Basic COCOMO co-efficients

Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size (SS)} = \frac{E}{D} \text{ Persons.}$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity (P)} = \frac{\text{KLOC}}{E} \text{ KLOC/PM.}$$

With the basic model, the software estimator has a useful tool for estimating quickly, by two runs on a pocket calculator, the cost and development time of a software project, once the size is estimated. The software estimator will have to assess by himself/herself which mode is the most appropriate.

Example 4.5

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes *i.e.*, organic, semidetached and embedded.

Solution

The basic COCOMO equations take the form:

$$E = a_b(\text{KLOC})^{b_b}$$

$$D = c_b(\text{KLOC})^{d_b}$$

Estimated size of the project = 400 KLOC

(*i*) **Organic mode**

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$
$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ M}$$

(*ii*) **Semidetached mode**

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$
$$D = 2.5(2462.79)^{.35} = 38.45 \text{ M}$$

(*iii*) **Embedded mode**

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$
$$D = 2.5(4772.8)^{.32} = 38 \text{ M}$$

As we have seen, effort calculated for embedded mode is approximately 4 times, the effort for organic mode. However, the effort calculated for semidetached mode is 2 times the effort of organic mode. There is a large difference in these values. But, surprisingly, the development time is approximately the same for all three modes. It is clear from here that the

selection of mode is very important. Since, development time is approximately the same, the only varying parameter is the requirement of persons. Every mode will have different manpower requirement. If we look to the Table 4.4, it is mentioned that for over 300 KLOC projects, embedded mode is the right choice. The selection of a mode is not only dependent on project size, but also on other parameters as mentioned in Table 4.4. We should be utmost careful about the selection of mode for the project.

Example 4.6

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

Solution

The semi-detached mode is the most appropriate mode; keeping in view the size, schedules and experience of the development team.

Hence $E = 3.0(200)^{1.12} = 1133.12$ PM

$D = 2.5(1133.12)^{.35} = 29.3$ M

$$\text{Average staff size (SS)} = \frac{E}{D} \text{ Persons}$$

$$= \frac{1133.12}{29.3} = 38.67 \text{ Persons.}$$

$$\text{Productivity} = \frac{\text{KLOC}}{E} = \frac{200}{1133.12} = 0.1765 \text{ KLOC/PM}$$

$$P = 176 \text{ LOC/PM.}$$

4.4.2 Intermediate Model

The basic model allowed for a quick and rough estimate, but it resulted in a lack of accuracy. Boehm introduced an additional set of 15 predictors called cost drivers in the intermediate model to take account of the software development environment. Cost drivers are used to adjust the nominal cost of a project to the actual project environment, hence increasing the accuracy of the estimate.

The cost drivers are grouped into four categories:

1. Product attributes
   (*a*) Required software reliability (RELY)
   (*b*) Database size (DATA)
   (*c*) Product complexity (CPLX)
2. Computer attributes
   (*a*) Execution time constraint (TIME)
   (*b*) Main storage constraint (STOR)
   (*c*) Virtual machine volatility (VIRT)
   (*d*) Computer turnaround time (TURN)

3. Personnel attributes
   (a) Analyst capability (ACAP)
   (b) Application experience (AEXP)
   (c) Programmer capability (PCAP)
   (d) Virtual machine experience (VEXP)
   (e) Programming language experience (LEXP)
4. Project attributes
   (a) Modern programming practices (MODP)
   (b) Use of software tools (TOOL)
   (c) Required development schedule (SCED)

Each cost driver is rated for a given project environment. The rating uses a scale very low, low, nominal, high, very high, extra high which describes to what extent the cost driver applies to the project being estimated. Table 4.5 gives the multiplier values for the 15 cost drivers and their rating as provided by Boehm [BOEH81].

Table 4.5: Multiplier values for effort calculations

Cost Drivers	*RATINGS*					
	Very low	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
Product attributes						
RELY	0.75	0.88	1.00	1.15	1.40	–
DATA	–	0.94	1.00	1.08	1.16	–
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer attributes						
TIME	–	–	1.00	1.11	1.30	1.66
STOR	–	–	1.00	1.06	1.21	1.56
VIRT	–	0.87	1.00	1.15	1.30	–
TURN	–	0.87	1.00	1.07	1.15	–
Personnel attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	–
AEXP	1.29	1.13	1.00	0.91	0.82	–
PCAP	1.42	1.17	1.00	0.86	0.70	–
VEXP	1.21	1.10	1.00	0.90	–	–
LEXP	1.14	1.07	1.00	0.95	–	–
Project attributes						
MODP	1.24	1.10	1.00	0.91	0.82	–
TOOL	1.24	1.10	1.00	0.91	0.83	–
SCED	1.23	1.08	1.00	1.04	1.10	–

The multiplying factors for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF). Typical values for EAF range from 0.9 to 1.4.

The intermediate COCOMO equations take the form:

$$E = a_i (KLOC)^{b_i} * EAF \tag{4.10}$$

$$D = c_i (E)^{d_i} \tag{4.11}$$

The co-efficients a_i, b_i, c_i and d_i are given in Table 4.6.

Table 4.6: Co-efficients for intermediate COCOMO

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

4.4.3 Detailed COCOMO Model

A large amount of work has been done by Boehm to capture all significant aspects of a software development. It offers a means for processing all the project characteristics to construct a software estimate. The detailed model introduces two more capabilities:

1. Phase-sensitive effort multipliers:

   Some phases (design, programming, integration/test) are more affected than others by factors defined by the cost drivers. The detailed model provides a set of phase sensitive effort multipliers for each cost driver. This helps in determining the manpower allocation for each phase of the project.

2. Three-level product hierarchy:

   Three product levels are defined. These are module, subsystem and system levels. The ratings of the cost drivers are done at appropriate level; that is, the level at which it is most susceptible to variation.

Development phases

A software development is carried out in four successive phases: plans/requirements, product design, programming and integration/test.

1. **Plan/requirements:** This is the first phase of the development cycle. The requirement is analyzed, the product plan is set up and a full product specification is generated. This phase consumes from 6% to 8% of the effort and 10% to 40% of the development time. These percentages depend not only on mode (organic, semi-detached or embedded), but also on the size.
2. **Product design:** The second phase of the COCOMO development cycle is concerned with the determination of the product architecture and the specification of the subsystem. This phase requires from 16% to 18% the nominal effort and can last from 19% to 38% of the development time.

3. **Programming:** The third phase of the COCOMO development cycle is divided into two sub phases: detailed design and code/unit test. This phase requires from 48% to 68% of the effort and lasts from 24% to 64% of the development time.
4. **Integration/Test:** This phase of the COCOMO development cycle occurs before delivery. This mainly consists of putting the tested parts together and then testing the final product. This phase requires from 16% to 34% of the nominal effort and can last from 18% to 34% of the development time.

Principle of the effort estimate

Size equivalent: As the software might be partly developed from software already existing (that is, re-usable code), a full development is not always required. In such cases, the parts of design document (DD%), code (C%) and integration (I%) to be modified are estimated. Then, an adjustment factor, A, is calculated by means of the following equation.

$$A = 0.4\ DD + 0.3\ C + 0.3\ I \tag{4.12}$$

The size equivalent is obtained by

$$S(\text{equivalent}) = (S \times A)/100 \tag{4.13}$$

where S represents the thousands of lines of code (KLOC) of the module. Multipliers have been developed that can be applied to the total project effort, E, and total project development time, D in order to allocate effort and schedule components to each phase in the life cycle of a software development program. There are assumed to be five distinct life cycle phases, and the effort and schedule for each phase are assumed to be given in terms of the overall effort and schedule by

$$E_p = \mu_p E \tag{4.14}$$

$$D_p = \tau_p D \tag{4.15}$$

where μ_p and τ_p are given in Table 4.7. There exist more sophisticated versions of this development that result in multipliers μ_p and τ_p that not only depend on the particular phase of the life cycle and mode of operation of the software but also contain the correction terms for the 15 attributes [BOEH81].

The COCOMO model is certainly the most thoroughly documented model currently available. It is very easy to use. And by doing so, the software manager can learn a lot about productivity, particularly from the very clear presentation of the cost drivers. The size and cost drivers can be progressively adjusted to realistic values to some extent. However, mode choice offers some difficulties, since it is not always possible to be sure which of the three modes is appropriate for a given software development—it might be a mixed mode.

Table 4.7: Effort and schedule fractions occurring in each phase of the lifecycle [SAGE90]

Mode & code size	*Plan & requirement*	*System design*	*Detail design*	*Module code & test*	*Integration and test*
Lifecycle Phase value of μ_p					
Organic Small S≈2	0.06	0.16	0.26	0.42	0.16
Organic Medium S≈32	0.06	0.16	0.24	0.38	0.22
Semidetached Medium S≈32	0.07	0.17	0.25	0.33	0.25
Semidetached Large S≈128	0.07	0.17	0.24	0.31	0.28
Embedded Large S≈128	0.08	0.18	0.25	0.26	0.31
Embedded Extra Large S≈320	0.08	0.18	0.24	0.24	0.34
Lifecycle Phase value of τ_p					
Organic Small S≈2	0.10	0.19	0.24	0.39	0.18
Organic Medium S≈32	0.12	0.19	0.21	0.34	0.26
Semidetached Medium S≈32	0.20	0.26	0.21	0.27	0.26
Semidetached Large S≈128	0.22	0.27	0.19	0.25	0.29
Embedded Large S≈128	0.36	0.36	0.18	0.18	0.28
Embedded Extra Large S≈320	0.40	0.38	0.16	0.16	0.30

There are five phases of software life cycle in Table 4.7. However, plan and requirement phase has been combined with system design and known as requirement and product design. Both include the most conceptual part of the life cycle. So the effort and time shown in Table 4.7 for plan and requirements phase are over and above the estimated effort and time. The actual distribution may start from system design phase. Hence, four phases are:

1. Requirement and product design
   (*a*) Plans and requirements
   (*b*) System design
2. Detailed Design
   (*a*) Detailed design
3. Code & Unit test
   (*a*) Module code & test
4. Integrate and Test
   (*a*) Integrate & Test.

COCOMO is highly calibrated model, based on previous experience. It is easy to use and documented properly. Actual data gathered from previous projects may help to determine the values of the constants of the model (like a, b, c and d). These values may very from organisation to organisation. However, this model ignores software safety & security issues. It also ignores many hardware and customer related issues. It is silent about the involvement and responsiveness of customer.

It does not give proper importance to software requirements and specification phase which has identified as the most sensitive phase of software development life cycle.

Example 4.7

A new project with estimated 400 KLOC embedded system has to be developed. Project manager has a choice of hiring from two pools of developers: Very highly capable with very little experience in the programming language being used or developers of low quality but a lot of experience with the programming language. What is the impact of hiring all developers from one or the other pool ?

Solution

This is the case of embedded mode and model is intermediate COCOMO.

Hence $E = a_i(\text{KLOC})^{d_i}$

$= 2.8(400)^{1.20} = 3712$ PM

Case I: Developers are very highly capable with very little experience in the programming being used.

$$\text{EAF} = 0.82 \times 1.14 = 0.9348$$

$$E = 3712 \times .9348 = 3470 \text{ PM}$$

$$D = 2.5(3470)^{0.32} = 33.9 \text{ M}$$

Case II: Developers are of low quality but lot of experience with the programming language being used.

$$\text{EAF} = 1.29 \times 0.95 = 1.22$$

$$E = 3712 \times 1.22 = 4528 \text{ PM}$$

$$D = 2.5(4528)^{0.32} = 36.9 \text{ M}$$

Case II requires more effort and time. Hence, low quality developers with lot of programming language experience could not match with the performance of very highly capable developers with very little experience.

Example 4.8

Consider a project to develop a full screen editor. The major components identified are (1) Screen Edit (2) Command language Interpreter (3) File input and output, (4) Cursor movement and (5) Screen movement. The sizes for these are estimated to be 4K, 2K, 1K, 2K and 3K delivered source code lines. Use COCOMO model to determine:

(*a*) Overall cost and schedule estimates (assume values for different cost drivers, with at least three of them being different from 1.0).

(*b*) Cost and Schedule estimates for different phases.

Solution

Size of five modules are:

Screen edit	= 4 KLOC
Command language interpreter	= 2 KLOC
File input and output	= 1 KLOC
Cursor movement	= 2 KLOC

Screen movement = 3 KLOC

Total **= 12 KLOC**

Let us assume that significant cost drivers are

(*i*) Required software reliability is high, *i.e.*, 1.15

(*ii*) Product complexity is high, *i.e.*, 1.15

(*iii*) Analyst capability is high, *i.e.*, 0.86

(*iv*) Programming language experience is low, *i.e.*, 1.07

(*v*) All other drivers are nominal.

$$EAF = 1.15 \times 1.15 \times 0.86 \times 1.07 = 1.2169$$

(*a*) The initial effort estimate for the project is obtained from the following equation

$$E = a_i (KLOC)^{b_i} \times EAF$$
$$= 3.2(12)^{1.05} \times 1.2169 = 52.91 \text{ PM}$$

Development time $D = C_i (E)^{d_i}$

$$= 2.5\,(52.91)^{0.38} = 11.29 \text{ M}$$

(*b*) Using the following equations and referring Table 4.7, phase wise cost and schedule estimates can be calculated.

$$E_p = \mu_p E$$
$$D_p = \tau_p D$$

Since size is only 12 KLOC, it is an organic small model. Phase wise effort distribution is given below:

System Design $= 0.16 \times 52.91 = 8.465$ PM

Detailed Design $= 0.26 \times 52.91 = 13.756$ PM

Module Code & Test $= 0.42 \times 52.91 = 22.222$ PM

Integration & Test $= 0.16 \times 52.91 = 8.465$ PM

Now Phase wise development time duration is

System Design $= 0.19 \times 11.29 = 2.145$ M

Detailed Design $= 0.24 \times 11.29 = 2.709$ M

Module Code & Test $= 0.39 \times 11.29 = 4.403$ M

Integration & Test $= 0.18 \times 11.29 = 2.032$ M

4.5 COCOMO-II

COCOMO-II is the revised version of the original COCOMO (discussed in article 4.4) and is developed at University of Southern California under the leadership of Dr. Barry Boehm. The model is tuned to the life cycle practices of the 21st century. It also provides a quantitative analytic framework, and set of tools and techniques for evaluating the effects of software technology improvements on software life cycle costs and schedules. The following categories of applications/projects are identified by COCOMO-II for the estimation [UCSD01] and are shown in Fig. 4.4.

(*i*) **End user programming:** This category is applicable to small systems, developed by end user using application generators. Some application generators are spreadsheets extended query system, report generators etc. End user may write small programs using application generators. The end users may not have sufficient knowledge about computers and software engineering practices. However, they may have in-depth knowledge about their business needs and practices. Hence, this excellent domain knowledge may motivate them to develop an application using user friendly tools like MS-Excel, MS-Access, MS-Studio etc.

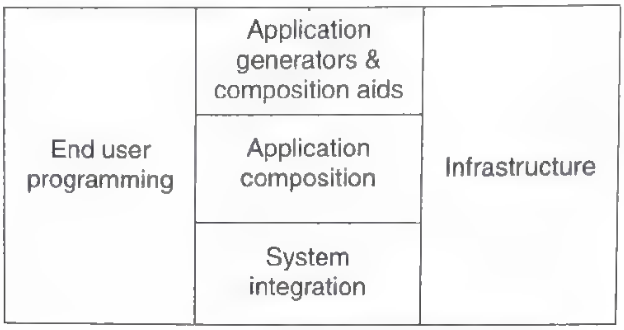


Fig. 4.4: Categories of applications/projects

(*ii*) **Infrastructure sector:** This category is applicable to the infrastructure development *i.e.*, the software that provides infrastructure like operating systems, database management systems, user interface management system, networking system etc. Some commercial examples are Microsoft products, Oracle, DB2, MAYA; LINUX, 3D-STUDIO.

Infrastructure developers generally have good knowledge of software development and software engineering practices and relatively little knowledge about applications. The product lines will have many reusable components, but the pace of technology (new processor, memory, communications, display, and multimedia technology) will require them to build many components and capabilities from scratch.

(*iii*) **Intermediate sectors:** This category is partitioned in three sub categories as shown in Fig. 4.4. Software developers will need to have good knowledge of software development and software engineering practices, experience and expertise of infrastructure software and indepth domain knowledge of one or more applications. Creating this talent pool is a major challenge.

Application generators and composition aids : This subcategory will create largely prepackaged capabilities for user programming. Typical firms operating in this sector are Microsoft, Lotus, Novell, Borland, Alias Wavefront, Oracle, IBM. Their product lines will have many reusable components, but also will require a good deal of new capability development from the scratch. Application composition aids will be developed both by the firms above and by software product line investments of firms in the application composition sector.

Application composition sector : This subcategory deals with applications which are too diversified to be handled by prepackaged solutions, but which are sufficiently simple to be rapidly composable from interoperable components. Typical components will be graphic user interface (GUI) builders, databases or object managers, domain specific components such as financial, medical, or industrial process control packages etc.

These applications are complex, large, versatile, diversified and require specialised developers with sound knowledge of development and software engineering practices.

However, they are developed using application generator environment like CASE tools, DBMS (DB2, oracle etc.), and 4GL programming tools (Developer 2000, Visual basic, Power builder, ASP, JSP, PHP etc).

System integration : This subcategory deals with large scale, highly embedded, or unprecedented systems. Portions of these systems can be developed with application composition capabilities, but their demands generally require a significant amount of upfront systems engineering and customised software development activities.

Stages of COCOMO-II : The end user programming sector does not need a COCOMO-II model. Its applications are normally developed in hours to days. Hence a simple activity based estimate will generally be sufficient.

COCOMO-II includes three stages. Stage I supports estimation of prototyping or application composition types of projects. Stage II supports estimation in the early design stage of a project, when less is known about the project's cost drivers. Stage III supports estimation in the Post-Architecture stage of a project. The details are given in Table 4.8.

Table 4.8: Stages of COCOMO-II

Stage No.	*Model name*	*Applicable for the types of projects*	*Applications*
Stage I	Application composition estimation model	Application composition	In addition to application composition type of projects, this model is also used for prototyping (if any) stage of application generators, infrastructure & system integration.
Stage II	Early design estimation model	Application generators, infrastructure & system integration.	Used in early design stage of a project, when less is known about the project.
State III	Post architecture estimation model	Application generators, infrastructure & system integration	Used after the completion of the detailed architecture of the project

4.5.1 Application Composition Estimation Model

The model is designed for quickly developed applications using interoperable components. Example of these components based systems are Graphic User Interface (GUI) builders, database or object managers, hypermedia handlers, smart data finders, and domain specific components such as financial, medical, or industrial process control packages. The model can also be used for the prototyping phase of application generator development, infrastructure sector and system integration projects.

In this model, size is first estimated using object points. The object points are easy to identify and count. The object in object points defines screens, reports, and 3GL modules as objects. This may or may not have any relationship to other definitions of "objects", such as those processing features like class affiliation, inheritance, encapsulation, message passing, and so forth [UCSDOI].

Object point estimation is a relatively new size estimation technique, but it is well suited in application composition sector. It is also a good match to associated prototyping efforts, based on the use of a rapid composition Integrated Computer Aided Software Engineering (ICASE) Environment providing graphic user interface builders, software development tools, and large, composable infrastructure and applications components. The steps required for the estimation of effort in Person-months are given in Fig. 4.5.

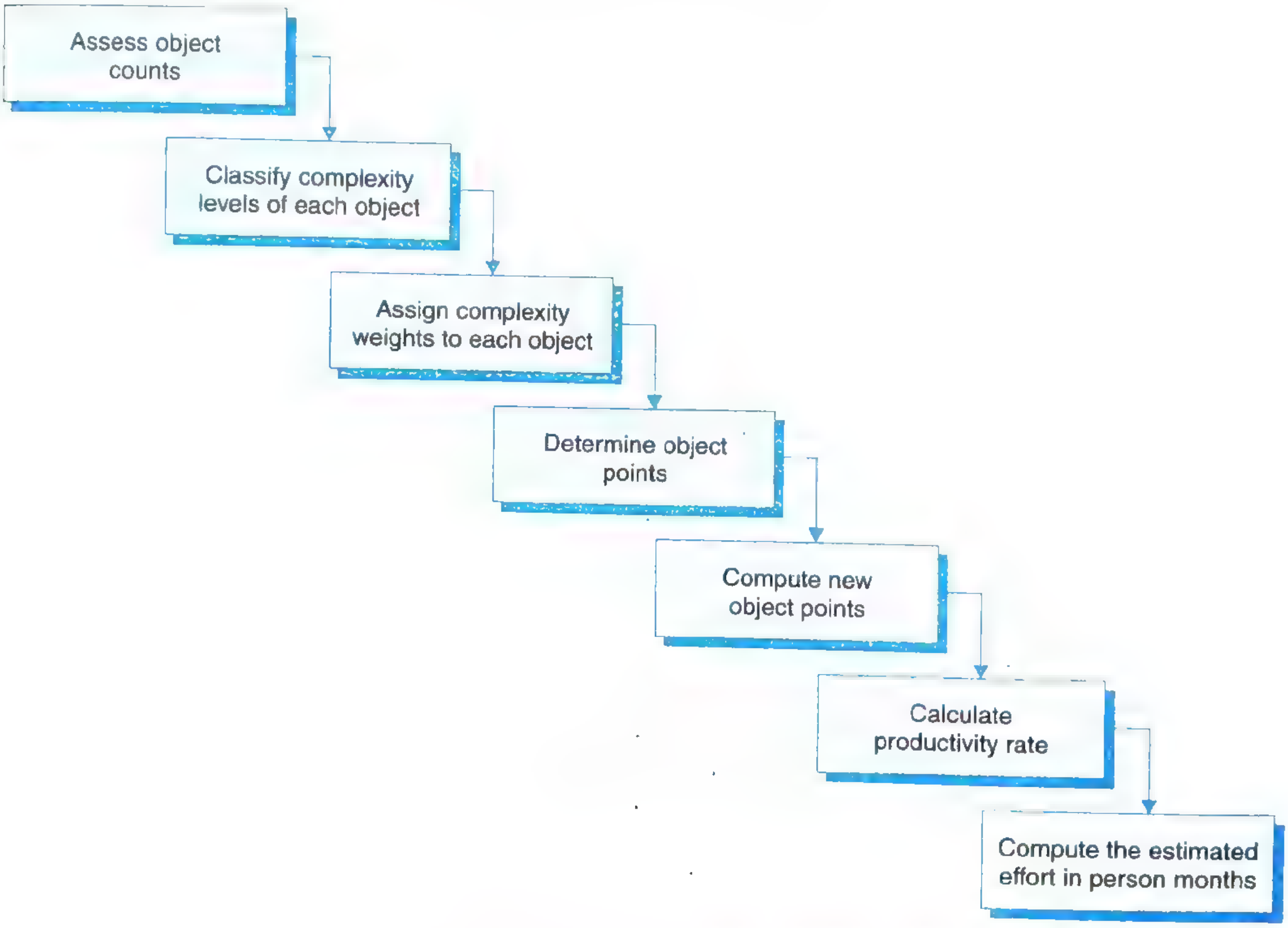


Fig. 4.5: Steps for the estimation of effort in person months

(*i*) **Assess object counts:** Estimate the number of screens, reports, and 3GL components that will comprise this application.

(*ii*) **Classification of complexity levels:** We have to classify each object instance into simple, medium and difficult complexity levels depending on values of its characteristics.

The screens are classified on the basis of number of views and sources and reports are on the basis of number of sections and sources. The details are given in Table 4.9.

Table 4.9(a): For screens

Number of views contained	# and sources of data tables		
	Total < 4 (< 2 server < 3 client)	Total < 8 (2 – 3 server 3 – 5 client)	Total 8 + (> 3 server, > 5 client)
< 3	Simple	Simple	Medium
3 – 7	Simple	Medium	Difficult
> 8	Medium	Difficult	Difficult

Table 4.9(b): For reports

Number of sections contained	# and sources of data tables		
	Total < 4 (< 2 server < 3 client)	Total < 8 (2 – 3 server 3 – 5 client)	Total 8 + (> 3 server, > 5 client)
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 +	Medium	Difficult	Difficult

(*iii*) **Assign comlexity weight to each object:** The weights are used for three object types *i.e.*, screen, report and 3GL components using the Table 4.10.

The weights reflect the relative effort required to implement an instance of that complexity level.

Table 4.10: Complexity weights for each level

Object type	Complexity weight		
	Simple	*Medium*	*Difficult*
Screen	1	2	3
Report	2	5	8
3GL Component	—	—	10

(*iv*) **Determine object points:** Add all the weighted object instances to get one number and this number is known as object-point count.

(*v*) **Compute new object points:** We have to estimate the percentage of reuse to be achieved in a project. Depending on the percentage reuse, the new object points (NOP) are computed.

$$\text{NOP} = \frac{(\text{object points}) * (100 - \%\ \text{reuse})}{100}$$

NOPs are the object points that will need to be developed and differ from the object point count because there may be reuse.

(*vi*) **Calculation of productivity rate:** The productivity rate can be calculated as:

Productivity rate (PROD) = NOP/Person month.

PROD values Calculation requires NOP and total person-months of past projects in similar environments. COCOMO-II application composition model gives the following Table 4.11 containing the values of PROD based on their data and experience.

Table 4.11: Productivity values

Developer's experience & capability; ICASE maturity & capability	*PROD (NOP/PM)*
Very low	4
Low	7
Nominal	13
High	25
Very high	50

(*vii*) **Compute the effort in person-months:** When PROD is known, we may estimate effort in Person-Months as:

$$\text{Effort in PM} = \frac{\text{NOP}}{\text{PROD}}$$

Example 4.9

Consider a database application project with the following characteristics:

(*i*) The application has 4 screens with 4 views each and 7 data tables for 3 servers and 4 clients.

(*ii*) The application may generate two report of 6 sections each from 07 data tables for two server and 3 clients. There is 10% reuse of object points.

The developer's experience and capability in the similar environment is low. The maturity of organisation in terms of capability is also low. Calculate the object point count, New object points and effort to develop such a project.

Solution

This project comes under the category of application composition estimation model.

Number of screens = 4 with 4 views each

Number of reports = 2 with 6 sections each.

From Table 4.9, we know that each screen will be of medium complexity and each report will be of difficult complexity.

Using Table 4.10 of complexity weights, we may calculate object point count

$$= 4 \times 2 + 2 \times 8 = 24$$

$$\text{NOP} = \frac{24 * (100 - 10)}{100} = 21.6$$

Table 4.11 gives the low value of productivity (PROD) *i.e.*, 7.

$$\text{Efforts in PM} = \frac{\text{NOP}}{\text{PROD}}$$

$$\text{Effort} = \frac{21.6}{7} = 3.086 \text{ PM}$$

4.5.2 The Early Design Model

The COCOMO-II models use the base equation of the form

$$PM_{nominal} = A * (size)^B$$

where $PM_{nominal}$ = Effort of the project in person months.

A = Constant representing the nominal productivity, provisionally set to 2.5

B = Scale factor

Size = Software size

The early design model uses Unadjusted Function Points (UFP) as the measure of size. This model is used in the early stages of a software project when very little may be known about the size of the product to be developed, the nature of the target platform, the nature of the personnel to be involved in the project or the detailed specifics of the process to be used. This model can be used in either Application Generator, System Integration, or Infrastructure Development Sector.

If B = 1.0, there is a linear relationship of effort and size. If value of B is not 1, there will be a non-linear relationship between size and effort. If B < 1.0, the rate of increase of effort decreases as the size of the product increases. If the product's size is doubled, the project effort is less than doubled.

If B > 1.0, the rate of increase of effort increases as the size of the product increases. This is due to the growth of interpersonal communications overheads and growth of large system integration overhead. Application composition model assumes the value of B to be 1. But the early design model assumes the value of B to be greater than 1. Thus, the basic assumption is that the effort spent in a project usually increases faster than the size of the project. The value of B is computed on the basis of scaling factors (or drivers) that may cause drop in productivity with increase in size.

Table 4.12: Scaling factors required for the calculation of the value of B

Scale factor	*Explanation*	*Remarks*
Precedentness	Reflects the previous experience on similar projects. This is applicable to individuals & organisation both in terms of expertise & experience.	Very low means no previous experiences, Extra high means that organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process.	Very low means a well defined process is used. Extra high means that the client gives only general goals.
Architecture/Risk resolution	Reflects the degree of risk analysis carried out.	Very low means very little analysis and Extra high means complete and thorough risk analysis.

(Contd...)

Team cohesion	Reflects the team management skills.	Very low means very little interaction & hardly any relationship among team members; Extra high means an integrated & effective team.
Process maturity	Reflects the process maturity of the organisation. Thus it is dependent on SEI-CMM level of the organisation.	Very low means organisation has no level at all and extra high means organisation is rated as highest level of SEI-CMM.

The scaling factors that COCOMO-II uses for the calculation of B are Precedentness, Development Flexibility, Architecture/Risk Resolution, Team Cohesion and Process Maturity. The details are given in Table 4.12. These factors are rated on a six point scale *i.e.*, very low, low, nominal, high, very high and extra high and are given in Table 4.13.

Table 4.13: Data for the Computation of B

Scaling factors	*Very low*	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
Precedentness	6.20	4.96	3.72	2.48	1.24	0.00
Development flexibility	5.07	4.05	3.04	2.03	1.01	0.00
Architecture/Risk resolution	7.07	5.65	4.24	2.83	1.41	0.00
Team cohesion	5.48	4.38	3.29	2.19	1.10	0.00
Process maturity	7.80	6.24	4.68	3.12	1.56	0.00

The value of B can be calculated as:

$$B = 0.91 + 0.01 * (\text{Sum of rating on scaling factors for the project}).$$

When all the scaling factors of a project are rated as extra high, the best value of B is obtained and is equal to 0.91. When all the scaling factors are very low, the worst value of B is obtained and is equal to 1.23. Hence value of B may vary from 0.91 to 1.23.

Early design cost drivers

There are seven early design cost drivers and are given below:

(*i*) Product Reliability and Complexity (RCPX)

(*ii*) Required Reuse (RUSE)

(*iii*) Platform Difficulty (PDIF)

(*iv*) Personnel Capability (PERS)

(*v*) Personnel Experience (PREX)

(*vi*) Facilities (FCIL)

(*vii*) Schedule (SCED)

Post architecture cost drivers

There are 17 cost drivers in the Post Architecture model. These are rated on a scale of 1 to 6 as given below:

[illegible] Low	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
1	2	3	4	5	6

The list of seventeen cost drivers is given below:

1. Reliability Required (RELY)
2. Database Size (DATA)
3. Product Complexity (CPLX)
4. Required Reusability (RUSE)
5. Documentation (DOCU)
6. Execution Time Constraint (TIME)
7. Main Storage Constraint (STOR)
8. Platform Volatility (PVOL)
9. Analyst Capability (ACAP)
10. Programmers Capability (PCAP)
11. Personnel Continuity (PCON)
12. Analyst Experience (AEXP)
13. Programmer Experience (PEXP)
14. Language & Tool Experience (LTEX)
15. Use of Software Tools (TOOL)
16. Site Locations & Communication Technology between Sites (SITE)
17. Schedule (SCED)

Mapping of early design cost drivers and post architecture cost drivers

The 17 Post Architecture Cost Drivers are mapped to 7 Early Design Cost Drivers and are given in Table 4.14. This mapping is essential because many parameters will not be known correctly in early design phase. The mapping combines estimated parameters in order to have reasonable view of cost drivers. In Post Architecture, all 17 drivers will be known with reasonable accuracy, hence no mapping is required.

Table 4.14: Mapping table

Early design cost drivers	*counter part combined post architecture cost drivers*
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PERS	ACAP, PCAP, PCON
PREX	AEXP, PEXP, LTEX
FCIL	TOOL, SITE
SCED	SCED

Product of cost drivers for early design model

The combined early design cost drivers may be obtained by summing the numerical values of the contributing Post Architecture cost drivers. The resulting totals are allocated to an expanded early design model rating scale from Extra Low to Extra High. The early design model rating scales always have a Nominal total equal to the sum of the Nominal ratings of its contributing Post-Architecture Cost drivers.

(i) **Product reliability and complexity (RCPX):** The cost driver combines four Post-Architecture cost drivers which are RELY, DATA, CPLX and DOCU. Here RELY & DOCU range from Very Low to Very High. DATA ranges from Low to Very High; and CPLX ranges from Very Low to Extra High. The numerical sum of their ratings thus ranges from 5(VL, L, VL, VL) to 21(VH, VH, EH, VH). For details please refer to Table 4.16. The RCPX rating levels are given below:

RCPX	*Extra low*	*Very low*	*Low*	*Nominal*	*High*	*Very high*	***Extra high***
Sum of RELY, DATA, CPLX, DOCU ratings	5, 6	7, 8	9-11	12	13-15	16-18	19-21
Emphasis on reliability, documentation	Very Little	Little	Some	Basic	Strong	Very Strong	Extreme
Product complexity	Very Simple	Simple	Some	Moderate	Complex	Very Complex	Extremely Complex
Database size	Small	Small	Small	Moderate	Large	Very Large	Very Large

(*ii*) **Required reuse (RUSE):** This early design model cost driver is same as its Post-architecture Counterpart. The RUSE rating levels are (As per Table 4.16):

	Vary low	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
	1	2	3	4	5	6
RUSE		None	Across project	Across program	Across product line	Across multiple product line

(*iii*) **Platform difficulty (PDIF):** This cost driver combines TIME, STOR, and PVOL of Post-Architecture cost drivers. From the Table 4.16 it is clear that TIME and STOR range from Nominal to Extra High; PVOL ranges from Low to Very High. The numerical sum of ratings thus ranges from 8(N, N, L) to 17 (EH, EH, VH). Hence PDIF rating levels are:

PDIF	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
Sum of Time, STOR & PVOL ratings	8	9	10-12	13-15	16-17
Time & storage constraint	≤ 50%	≤ 50%	65%	80%	90%
Platform Volatility	Very stable	Stable	Somewhat stable	Volatile	Highly volatile

(*iv*) **Personnel capability (PERS):** This cost driver combines three Post-Architecture Cost drivers. These drivers are analyst capability (ACAP), Programmers Capability (PCAP) and Personnel Continuity (PCON). Each of these has a rating scale from Very Low to very High as per Table 4.16. Adding up their numerical ratings produces values ranging from 3 to 15. PERS rating levels are calculated with the help of Table 4.16 and are given below:

PERS	*Extra low*	*Very low*	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
Sum of ACAP, PCAP, PCON ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Combined ACAP & PCAP Percentile	20%	39%	45%	55%	65%	75%	85%
Annual Personnel Turnover	45%	30%	20%	12%	9%	5%	4%

(*v*) **Personnel experience (PREX):** This early design cost driver combines three Post Architecture Cost drivers, which are: application experience (AEXP), platform experience (PEXP) and language and Tool experience (LTEX). Each of these range from Very Low to Very High and numerical sum of their ratings ranges from 3 to 15. The PREX rating levels are obtained using Table 4.16 and are given below:

PREX	*Extra low*	*Very low*	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
Sum of AEXP, PEXP and LTEX ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Applications, Platform, Language & Tool Experience	≤ 3 months	5 months	9 months	1 year	2 year	4 year	6 year

(*vi*) **Facilities (FCIL):** This depends on two Post Architecture Cost drivers: Use of Software Tools (TOOL) and multisite development (SITE). TOOL ranges from Very Low to Very High; SITE ranges from Very Low to Extra High. Thus the numeric sum of their rating ranges from 2(VL, VL) to 11(VH, EH). FCIL rating levels are obtained using Table 4.16.

FCIL	*Extra low*	*Very low*	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
Sum of TOOL & SITE ratings	2	3	4, 5	6	7, 8	9, 10	11
Tool support	Minimal	Some	Simple CASE tools	Basic life cycle tools	Good support of tools	Very strong use of tools	Very strong & well integrated tools
Multisite conditions development support	Weak support of complex multisite development	Some support	Moderate support	Basic support	Strong support	Very strong support	Very strong support

(*vii*) **Schedule (SCED):** The early design cost driver is the same as Post Architecture Counter part and rating levels are given below using Table 4.16.

SCED	*Very Low*	*Low*	*Nominal*	*High*	*Very High*
Schedule	75% of Nominal	85%	100%	130%	160%

The seven early design cost drivers have been converted into numeric values with a Nominal value 1.0. These values are used for the calculation of a factor called "Effort multiplier" which is the product of all seven early design cost drivers. The numeric values are given in Table 4.15.

Table 4.15: Early design Parameters

Early design cost drivers	*Extra low*	*Very low*	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
RCPX	.73	.81	.98	1.0	1.30	1.74	2.38
RUSE	—	—	0.95	1.0	1.07	1.15	1.24
PDIF	—	—	0.87	1.0	1.29	1.81	2.61
PERS	2.12	1.62	1.26	1.0	0.83	0.63	0.50
PREX	1.59	1.33	1.12	1.0	0.87	0.71	0.62
FCIL	1.43	1.30	1.10	1.0	0.87	0.73	0.62
SCED	—	1.43	1.14	1.0	1.0	1.0	—

The early design model adjusts the nominal effort using 7 effort multipliers (EMs). Each effort multiplier (also called cost drivers) has 7 possible weights as given in Table 4.15. These factors are used for the calculation of adjusted effort as given below:

$$PM_{adjusted} = PM_{nominal} \times \left[\prod_{i=1}^{7} EM_i\right]$$

$PM_{adjusted}$ effort may very even up to 400% from $PM_{nominal}$.

Hence $PM_{adjusted}$ is the fine tuned value of effort in the early design phase.

Example 4.10

A software project of application generator category with estimated 50 KLOC has to be developed. The scale factor (B) has low precedentness, high development flexibility and low team cohesion. Other factors are nominal. The early design cost drivers like platform difficult (PDIF) and Personnel Capability (PERS) are high and others are nominal. Calculate the effort in person months for the development of the project.

Solution

Here $B = 0.91 + 0.01 * (\text{Sum of rating on scaling factors for the project})$

$= 0.91 + 0.01 * (4.96 + 2.03 + 4.24 + 4.38 + 4.68)$

$= 0.91 + 0.01(20.29) = 1.1129$

$PM_{nominal} = A * (\text{size})^B$

$= 2.5 * (50)^{1.1129} = 194.41$ Person months.

The 7 cost drivers are

PDIF = high (1.29)
PERS = high (0.83)
RCPX = nominal (1.0)
RUSE = nominal (1.0)
PREX = nominal (1.0)
FCIL = nominal (1.0)
SCEO = nominal (1.0)

$$PM_{adjusted} = PM_{nominal} * \left[\prod_{i-1}^{7} EM_i\right]$$

$= 194.41 * [1.29 \times 0.83] = 194.41 \times 1.07$

$= 208.155$ Person-months.

4.5.3 Post Architecture Model

The Post architecture Model is the most detailed estimation model and is intended to be used when a software life cycle architecture has been completed. This model is used in the development and maintenance of software products in the application generators, system integration or infrastructure sectors.

The Post Architecture model adjusts nominal effort using 17 efforts multipliers. The large number of multipliers takes advantage of the greater knowledge available later in the development stage.

$$PM_{adjusted} = PM_{nominal} \times \left[\prod_{i=1}^{17} EM_i\right]$$

EM : Effort multiplier which is the product of 17 cost drivers.

The 17 cost drivers of the Post Architecture model are described in the Table 4.16.

Table 4.16: Post Architecture Cost Driver rating level summary

Cost driver	*Purpose*	*Very low*	*Low*	*Nominal*	*High*	*Very high*	***Extra high***
RELY (Reliability required)	Measure of the extent to which the software must perform its intended function over a period of time	Only slight inconvenience	Low, easily recoverable losses	Moderate, easily recoverable losses	High financial loss	Risk to human life	—
DATA (Data base size)	Measure the affect of large data requirements on product development	—	$\frac{\text{Database size(D)}}{\text{Prog. size (P)}} < 10$	$10 \le \frac{D}{P} < 100$	$100 \le \frac{D}{P} < 1000$	$\frac{D}{P} \ge 1000$	—
CPLX (Product complexity)	Complexity is divided into five areas: Control operations, computational operations, device dependent operations, data management operations & User Interface management operations.	See Table 4.17					
RUSE	Measure the required reusability	—	None	Across project	Across program	Across product line	Across multiple product line
DOCU Documentation	Suitability of the project's documentation to its life cycle needs	Many life cycle needs uncovered	Some needs uncovered	Adequate	Excessive for life cycle needs	Very Excessive	—

(Contd...)

TIME (Execution Time constraint)	Measure of execution time constraint on software	—	—	≤ 50% use of a available execution time	70%	85%	95%
STOR (Main storage constraint)	Measure of main storage constraint on software	—	—	≤ 50% use of available storage	70%	85%	95%
PVOL (Platform Volatility)	Measure of changes to the OS, compilers, editors, DBMS etc.	—	Major changes every 12 months & minor changes every 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 week Minor: 2 days	—
ACAP (Analyst capability)	Should include analysis and design ability, efficiency & thoroughness, and communication skills.	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—
PCAP (Programmers capability)	Capability of Programmers as a team. It includes ability, efficiency, thoroughness & communication skills	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—
PCON (Personnel Continuity)	Rating is in terms of Project's annual personnel turnover	48%/year	24%/year	12%/year	6%/year	3%/year	—
AEXP (Applications Experience)	Rating is dependent on level of applications experience.	≤ 2 months	6 months	1 year	3 year	6 year	—
PEXP (Platform experience)	Measure of Platform experience	≤ 2 months	6 months	1 year	3 year	6 year	—

(Contd...)

LTEX (Language & Tool experience)	Rating is for Language & tool experience	≤ 2 months	6 months	1 year	3 year	6 year	—
TOOL (Use of software tools)	It is the indicator of usage of software tools	No use	Beginning to use	Some use	Good use	Routine & habitual use	—
SITE (Multisite development)	Site location & Communication technology between sites	International with some phone & mail facility	Multicity & multi company with individual phones, FAX	Multicity & multi company with Narrow band mail	Same city or Metro with wideband electronic communication	Same building or complex with wideband electronic communication & Video conferencing	Fully co-located with interactive multimedia
SCED (Required Development Schedule)	Measure of Schedule constraint. Ratings are defined in terms of percentage of schedule stretch-out or acceleration with respect to nominal schedule	75% of nominal	85%	100%	130%	160%	—

Product complexity is based on control operations, computational operations, device dependent operations, data management operations and user interface management operations. Module complexity ratings are given in Table 4.17.

The numeric values of these 17 cost drivers are given in Table 4.18 for the calculation of the product of efforts *i.e.*, effort multiplier (EM). Hence PM adjusted is calculated which will be a better and fine tuned value of effort in person months.

Table 4.17: Module complexity ratings

	Control operations	*Computational operations*	*Device-dependent operations*	*Data management operations*	*User interface management operations*
Very low	Straight-line code with a few non-nested structured programming operators: DOs, CASEs, IF THEN ELSEs. Simple module composition via procedure calls or simple scripts.	Evaluation of simple expressions : *e.g.*, $A = B + C*(D - E)$	Simple read, write statements with simple formats.	Simple arrays in main memory. Simple COTS-DB queries, updates.	Simple input forms, report generators.
Low	Straight forward nesting of structured programming operators. Mostly simple predicates	Evaluation of moderate-level expressions: *e.g.*, $D = SQRT(B**2-4*A*C)$	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level.	Single file subsetting with no data structure changes, no edits, no intermediate files. Moderately complex COTS-DB queries, updates.	Use of simple graphic user interface (GUI) builders.
Nominal	Mostly simple nesting. Some inter module control Decision tables. Simple callbacks or message passing, including middleware-supported distributed processing.	Use of standard maths and statistical routines. Basic matrix/vector operations.	I/O processing includes device selection, status checking and error processing.	Multi-file input and single file output. Simple structural changes, simple edits. Complex COTS-DB queries, updates.	Simple use of widget set.

(Contd.)...

High	Highly nested structured programming operators with many compound predicates. Queue and stack control. Homogeneous, distributed processing. Single processor soft real-time control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns.	Operations at physical I/O level (physical storage address translations; seeks, read, etc.) Optimized I/O overlap.	Simple triggers activated by data stream contents. Complex data restructuring.	Widget set development and extension. Simple voice I/O, multimedia.
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling. Task synchronization, complex callbacks, heterogeneous distributed processing. Single-processor hard real-time control.	Difficult but structured numerical analysis: near-singular matrix equations, partial differential equations. Simple parallelization.	Routines for interrupt diagnosis, servicing, masking. Communication line handling. Performance-intensive embedded systems.	Distributed database coordination. Complex triggers. Search optimization.	Moderately complex 2D/3D, dynamic graphics, multimedia.
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control. Distributed hard real-time control.	Difficult and unstructured numerical analysis: highly accurate analysis of noisy, stochastic data. Complex parallelization.	Device timing-dependent coding, micro-programmed operations. Performance-critical embedded systems.	Highly coupled, dynamic relational and object structures. Natural language data management.	Complex multimedia, virtual reality.

Table 4.18: 17 Cost drivers

Cost driver	*Rating*					
	Very low	*Low*	*Nominal*	*High*	*Very high*	*Extra high*
RELY	0.75	0.88	1.00	1.15	1.39	
DATA		0.93	1.00	1.09	1.19	
CPLX	0.75	0.88	1.00	1.15	1.30	1.66
RUSE		0.91	1.00	1.14	1.29	1.49
DOCU	0.89	0.95	1.00	1.06	1.13	
TIME			1.00	1.11	1.31	1.67
STOR			1.00	1.06	1.21	1.57
PVOL		0.87	1.00	1.15	1.30	
ACAP	1.50	1.22	1.00	0.83	0.67	
PCAP	1.37	1.16	1.00	0.87	0.74	
PCON	1.24	1.10	1.00	0.92	0.84	
AEXP	1.22	1.10	1.00	0.89	0.81	
PEXP	1.25	1.12	1.00	0.88	0.81	
LTEX	1.22	1.10	1.00	0.91	0.84	
TOOL	1.24	1.12	1.00	0.86	0.72	
SITE	1.25	1.10	1.00	0.92	0.84	0.78
SCED	1.29	1.10	1.00	1.00	1.00	

Schedule estimation

Development time can be calculated using $PM_{adjusted}$ as a key factor and the desired equation is:

$$TDEV_{nominal} = [\phi \times (PM_{adjusted})^{(0.28 + 0.2(B - 0.091))}] \quad * \frac{SCED\ \%}{100}$$

where ϕ = constant, provisionally set to 3.67

$TDEV_{nominal}$ = calendar time in months with a scheduled constraint

B = Scaling factor

$PM_{adjusted}$ = Estimated effort in Person months (after adjustment)

Size measurement

Size can be measured in any unit and the model can be calibrated accordingly. However, COCOMO II details are:

(*i*) Application composition model uses the size in object points.

(*ii*) The other two models use size in KLOC.

Early design model uses unadjusted function points. These function points are converted into KLOC using Table 4.19. Post architecture model may compute KLOC after defining LOC counting rules. If function points are used, then use unadjusted function points and convert it into KLOC using Table 4.19 [JUNE 91].

Table 4.19: Converting function points to lines of code

Language	*SLOC/UFP*
Ada	71
AI Shell	49
APL	32
Assembly	320
Assembly (Macro)	213
ANSI/Quick/Turbo Basic	64
Basic-Compiled	91
Basic-Interpreted	128
C	128
C++	29
ANSI Cobol 85	91
Fortan 77	105
Forth	64
Jovial	105
Lisp	64
Modula 2	80
Pascal	91
Prolog	64
Report Generator	80
Spreadsheet	6

COCOMO II reflects the experience and data collection of developers. It is a complex model and there are many attributes with too much scope for uncertainty in estimating their values. Each organisation should calibrate the model and attribute values according to its own historical data which may reflect local circumstances that affect the model.

Example 4.11

Consider the software project given in example 4.10. Size and scale factor (B) are the same. The identified 17 Cost drivers are high reliability (RELY), very high database size (DATA), high execution time constraint (TIME), very high analyst capability (ACAP), high programmers capability (PCAP). The other cost drivers are nominal. Calculate the effort in Person-Months for the development of the project.

Solution

Here $B = 1.1129$

$PM_{nominal} = 194.41$ Person-months

$$PM_{adjusted} = PM_{nominal} \times \left[\prod_{i=1}^{17} EM_i\right]$$

$$= 194.41 \times (1.15 \times 1.19 \times 1.11 \times 0.67 \times 0.87)$$

$$= 194.41 \times 0.885$$

$$= 172.05 \text{ Person-months.}$$

If analyst capability is very high alongwith high programmers capability, the effort will be reduced significantly. If such human resource factors are low, effort will be increased drastically. Therefore, human resource factors should be considered very carefully and are very important for the success of the project. If we consider estimated adjusted effort ($PM_{adjusted}$) of both the cases, the values are 208.155 Person-months and 172.05 Person-months. Hence difference is of 36.105 Person-months. More we know, better is the estimate, hence, effort estimated in Post Architecture model is more realistic and reasonable.

4.6 THE PUTNAM RESOURCE ALLOCATION MODEL

Norden [NORD58] of IBM observed that the Rayleigh curve can be used as an approximate model for a range of hardware development projects. This approach was later extended by Putnam to apply to software projects. Putnam observed that the Rayleigh curve (Fig. 4.6) was a close representation, not only at the project level but also for software subsystem development. As many as 150 projects were studied by Norden [NORD77] and subsequently by Putnam, and apparently both researchers observed the same tendency for the manpower curve to rise, peak, and then exponentially trail off as a function of time.

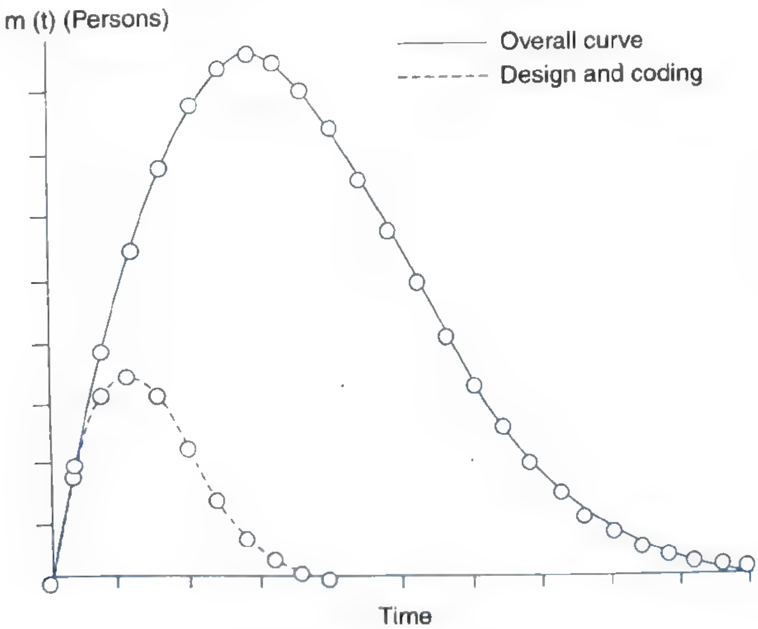


Fig. 4.6: The Rayleigh manpower loading curve.

4.6.1 The Norden/Rayleigh Curve

The Norden/Rayleigh equation represents manpower, measured in persons per unit time as a function of time. It is usually expressed in person-year/year (PY/YR). The Rayleigh curve is modeled by the differential equation

$$m(t) = \frac{dy}{dt} = 2Kat\,e^{-at^2} \tag{4.16}$$

where dy/dt is the manpower utilization rate per unit time, "t" is elapsed time, "a" is a parameter that affects the shape of the curve, and "K" is the area under the curve in the interval $[0, \infty]$. Integrating equation (4.16), on interval $[0, t]$, we obtain

$$y(t) = K[1 - e^{-at^2}] \tag{4.17}$$

where $y(t)$ is the cumulative manpower used upto time t.

$$y(0) = 0$$

$$y(\infty) = K$$

The cumulative manpower is null at the start of the project, and grows monotonically towards the total effort K (area under the curve).

It can be seen from the Fig. 4.7 that the parameter "a", which has the dimensions of $1/\text{time}^2$, plays an important role in the determination of the peak manpower. The larger the value of "a", earlier the peak time occurs and steeper is the person profile. By deriving the manpower function relative to time and finding the zero value of this derivative, the relationship between the peak time, "t_d", and "a" can be found to be:

$$\frac{d^2y}{dt^2} = 2Kae^{-at^2}\left[1 - 2at^2\right] = 0$$

$$\therefore \quad t_d^2 = \frac{1}{2a} \tag{4.18}$$

"t_d" denotes the time where maximum effort rate occurs. Thus, the point "t_d" on the time scale should correspond very closely to the total project development time. If we substitute "t_d" for t in equation (4.17), we can obtain an estimate for development time

$$E = y(t) = K\left(1 - e^{-\frac{t_d^2}{2t_d^2}}\right) = K(1 - e^{-0.5})$$

$$E = y(t) = 0.3935\ K \tag{4.19}$$

Actually if we divide the life cycle of a project into phases, each phase can be modeled by a curve of the form given in Fig. 4.6.

As shown in equation (4.18), the peak manning time is related to "a". Therefore, "a" can be obtained from the peak time as follows:

$$a = \frac{1}{2t_d^2}$$

The number of people involved in the project at the peak time then becomes easy to determine by replacing "a" with $1/2t_d^2$ in the Norden/ Rayleigh model. By making this substitution in equation (4.16), we have

$$m(t) = \frac{2K}{2t_d^2} te^{-\frac{t^2}{2t_d^2}}$$

$$= \frac{K}{t_d^2} te^{-\frac{t^2}{2t_d^2}}$$

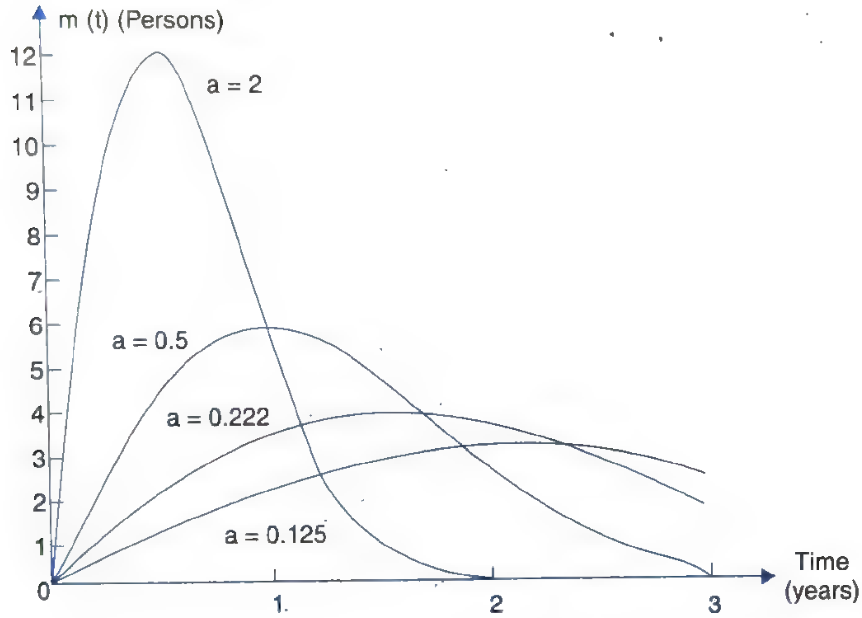


Fig. 4.7 Influence of parameter 'a' on the manpower distribution

At time $t = t_d$, the peak manning, $m(t_d)$ is obtained, which is denoted by m_o. Thus, expression for the peak manning of the project is

$$m_o = \frac{K}{t_d\sqrt{e}}$$

Where "K" is the total project cost (or effort) in person-years, "t_d" is the delivery time in years, "m_o" is the number of persons employed at the peak.

The average rate of software team build-up can also be calculated by dividing m_o by t_d.

Example 4.12

A software project is planned to cost 95 PY in a period of 1 year and 9 months. Calculate the peak manning and average rate of software team build up.

Solution

Software project cost	K = 95 PY
Peak development time	t_d = 1.75 years
Peak manning	$= m_o = \frac{K}{t_d\sqrt{e}}$

$$\frac{95}{1.75 \times 1.648} = 32.94 \cong 33 \text{ persons}$$

Average rate of software team build-up

$= m_o/t_d = 33/1.75 = 18.8$ person/year or 1.56 person/month.

Example 4.13

Consider a large-scale project for which the manpower requirement is K = 600 PY and the development time is 3 years 6 months.

(*a*) Calculate the peak manning and peak time.

(*b*) What is the manpower cost after 1 year and 2 months?

Solution

(*a*) We know t_d = 3 years and 6 months = 3.5 years

Now $$m_o = \frac{K}{t_d\sqrt{e}}$$

∴ $$m_o = 600/(3.5 \times 1.648) \cong 104 \text{ persons}$$

(*b*) We know

$$y(t) = K\,[1 - e^{-at^2}]$$

$$t = 1 \text{ year and 2 months} = 1.17 \text{ years}$$

$$a = \frac{1}{2t_d^2} = \frac{1}{2 \times (3.5)^2} = 0.041$$

$$y(1.17) = 600[1 - e^{-0.041(1.17)^2}] = 32.6 \text{ PY}$$

4.6.2 Difficulty Metric

The slope of the manpower distribution at start time (t = 0) also has some useful properties. By differentiating the Norden/Rayleigh function with respect to time, the following equation is obtained.

$$m'(t) = \frac{d^2y}{dt^2} = 2Kae^{-at^2}\,(1 - 2at^2)$$

Then, for $t = 0$,

$$m'(0) = 2Ka = \frac{2K}{2t_d^2} = \frac{K}{t_d^2} \tag{4.20}$$

The ratio $\frac{K}{t_d^2}$ is called difficulty and is denoted by D, which is measured in person/year:

$$D = \frac{K}{t_d^2} \tag{4.21}$$

This relationship shows that a project is more difficult to develop when the manpower demand is high or when the time schedule is short (small t_d). It is also interesting to note that difficult projects will tend to have a steeper demand for manpower at the beginning for the same time scale. After studying a large number (about 50) of Army developed software projects, Putnam observed that for systems that were relatively easy to develop, D tended to be small, while for systems that were relatively hard to develop, D tended to be large.

Peak manning is defined as

$$m_0 = \frac{K}{t_d \sqrt{e}}$$

We notice that the difficulty, D, is also related to the peak manning, "m_0" and the development time "t_d" by

$$D = \frac{K}{t_d^2} = \frac{m_0 \sqrt{e}}{t_d}$$

Thus, difficult projects tend to have a higher peak manning for a given development time, which is in line with Norden's observations relative to the parameter "a" [LOND87].

Manpower buildup

D is dependent upon "K" and "t_d". The derivative of D relative to "K" and "t_d" are:

$$D'(t_d) = \frac{-2K}{t_d^3} \text{ Person/years}^2$$

$$D'(K) = \frac{1}{t_d^2} \text{ year}^{-2}$$

In practice, $D^1(K)$ will always be very much smaller than the absolute value of $D'(t_d)$. This difference in sensitivity is shown by considering two projects

Project A : Cost = 20 PY & t_d = 1 year

Project B : Cost = 120 PY & t_d = 2.5 years

The derivative values are

Project A : $D'(t_d) = -40$ & $D'(K) = 1$

Project B : $D'(t_d) = -15.36$ & $D'(K) = 0.16$

This shows that a given software development is time sensitive.

Putnam also observed that the difficulty derivative relative to time played an important role in explaining the behaviour of software development. He noted that if the project scale is increased the development time also increases to such an extent that the quantity K/t_d^3 remains constant around a value, which could be 8, 15 or 27. This quantity is represented by D_0 and can be expressed as:

$$D_0 = \frac{K}{t_d^3} \text{ Person/year}^2$$

The value of D_0 is related to the nature of software developed in the following way:

- $D_0 = 8$ refers to entirely new software with many interfaces and interactions with other systems.
- $D_0 = 15$ refers to new stand alone system.
- $D_0 = 27$ refers to the software that is rebuilt from existing software.

Putnam also discovered that D_0 could vary slightly from one organisation to another depending on the average skill of the analysts, developers and the management involved.

In practice, D_0 has a strong influence on the shape of the manpower distribution. The larger D_0 is, the steeper manpower distribution is, and the faster the necessary manpower build up will be. For this reason, the quantity D_0 is called the manpower build up.

Example 4.14

Consider the example 4.13 and calculate the difficulty and manpower build up.

Solution

We know

Difficulty $$D = \frac{K}{t_d^2}$$

$$= \frac{600}{(3.5)^2} = 49 \text{ person/year}$$

Manpower build up can be calculated by following equation

$$D_0 = \frac{K}{t_d^3}$$

$$= \frac{600}{(3.5)^3} = 14 \text{ person/year}^2.$$

4.6.3 Productivity Versus Difficulty

It is appropriate to find relationship between productivity and difficulty. Productivity is defined as the number of lines of code developed per person-month. Putnam has observed that productivity is proportional to the difficulty

$$P \propto D^{\beta} \qquad (4.22)$$

The average productivity may be defined as :

P = Lines of code produced / Cumulative manpower used to produce code

$$P = S/E \qquad (4.23)$$

where S is lines of code produced and E is cumulative manpower used from $t = 0$ to $t = t_d$ (inception of the project to the delivery time).

Using non-linear regression, Putnam determined from an analysis of 50 army projects that

$$P = \phi\, D^{-2/3} \qquad (4.24)$$

Using equation 4.23, this relationship may be written as

$$S = \phi\, D^{-2/3} E$$

$$= \phi\, D^{-2/3} (0.3935\ K)$$

Using equation 4.21, we have

$$S = \phi\left[\frac{K}{t_d^2}\right]^{-\frac{2}{3}} K(0.3935)$$

$$S = 0.3935\ \phi K^{1/3} t_d^{4/3} \tag{4.25}$$

In the usual form of this expression, the quantity 0.3935 ϕ is replaced by a co-efficient C, which is given the name of Technology Factor. It reflects the effect of various factors on productivity such as hardware constraints, program complexity, personnel experience levels, and the programming environment. Putnam has proposed using a discrete spectrum of 20 values for C ranging from 610 to 57314 (assuming that K is measured in person-years and T in years) depending on an assessment of the technology factor that applies to the project under consideration. Equation 4.25 may be modified and now written as:

$$S = CK^{1/3} t_d^{4/3} \tag{4.26}$$

The value of C can also be found out as:

$$C = S.K^{-1/3} t_d^{-4/3} \tag{4.27}$$

It is easy to use the size, cost and development time of past projects to determine the value of C and hence to revise the value of C obtained to model forth coming projects.

4.6.4 The Trade-off between Time Versus Cost

In software projects, time cannot be freely exchanged against cost. Such a trade off is limited by the nature of software development. For a given organisation, developing a software of size S, the quantity obtained from equation 4.26 is constant. Using equation 4.26, we have

$$K^{1/3} t_d^{4/3} = S/C$$

If we raise power by 3, then Kt_d^4 is constant for a constant size software. A compression of the development time t_d will produce an increase of manpower cost. If compression is excessive, not only would the software development cost much more, but also the development would become so difficult that it would increase the risk of being unmanageable. This is in line with a remark made by Boehm that the time scale should never be reduced to less than 75% of its initial calculated value [LOND87].

The name given by Putnam to the later versions of this model is Software Life cycle Methodology (SLIM). This model is a combination of expertise and statistical computations and could be used effectively for predictive purposes if we had a suitable algorithm that we might use to predict the value of C for a software project. Using equation 4.27, we have

$$K = \frac{1}{t_d^4}\left[\frac{S}{C}\right]^3 \tag{4.28}$$

For a software product of a given size and fixed development environment, equation (4.28) implies that the effort K varies inversely as the fourth power of the development time. For instance, if we take the constant C to be 5000 and if we estimate the size of the project S = 500,000 LOC then

$$K = \frac{1}{t_d^4}(100)^3$$

Table 4.20 shows how the required effort in person-years changes as the development time measured in years changes. Thus, reducing the development time from 5 years to 4 years would increase the total effort and the cost by a factor 2.4, reducing it to 3 years would increase them by a factor of 7.7.

Table 4.20: Manpower versus development time

t_d (years)	K (person-years)
5.0	1600
4.0	3906
3.5	6664
3.0	12346

Putnam attempted to offer support for his use of "t_d^4" in equation (4.28) after examining 750 software systems [PUTN84]. With 251 of them it was shown that equation (4.28) is an acceptable model of the relationship among "K", "S", and "t_d". However C was computed using equation 4.27; it was not the result of some independent assessment of the technology level. Therefore, the data from 251 systems given in [PUTN84] may be said to offer only marginal support for the fourth power law. Furthermore, there was no evidence offered that the same relationship holds for the other 499 systems.

4.6.5 Development Sub-cycle

All that has been described so far is related to the project life cycle, as represented in Fig. 4.8, by the project curve.

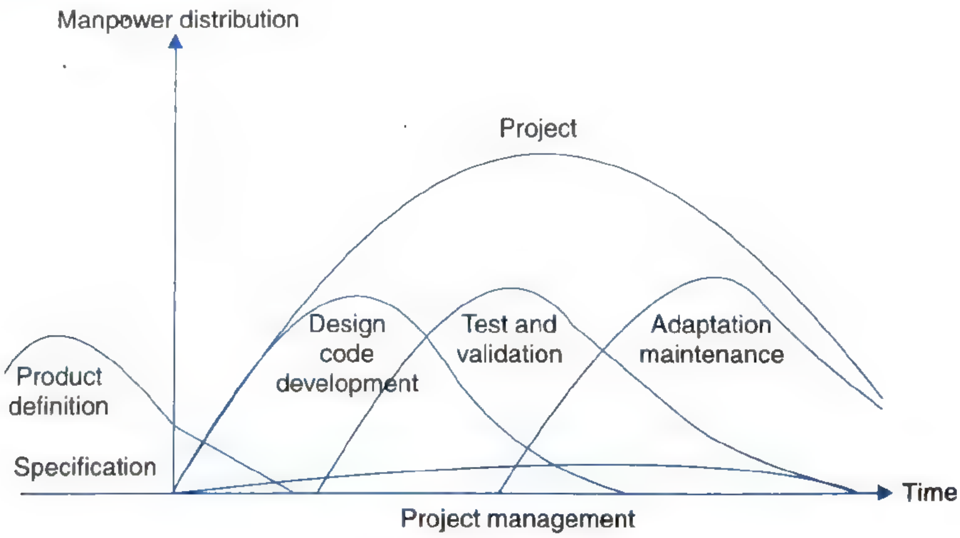


Fig. 4.8: Project life cycle

Curve is represented by a Rayleigh function, which gives the manning level relative to time and reaches a peak at time t_d. The project curve is the addition of two curves called development curve and test and validation curve. Both the curves are the sub-cycles of the project curve and can be modeled by Rayleigh function.

Let $m_d(t)$ and $y_d(t)$ be the design manning and the cumulative design manpower cost which can be represented by:

$$m_d(t) = 2K_d bt\, e^{-bt^2} \tag{4.29}$$

$$y_d(t) = K_d[1-e^{-bt^2}] \tag{4.30}$$

An examination of $m_d(t)$ function shows a non-zero value for m_d at time "t_d". This is because the manpower involved in design and coding is still completing this activity after "t_d" in the form of rework due to the validation of the product. Nevertheless, for the model a level of completion has to be assumed for development.

It is good practical assumption to assume that the development, will be 95% completed by the time t_d, this gives

$$\frac{y_d(t)}{K_d} = 1-e^{-bt^2} = 0.95 \tag{4.31}$$

It is then legitimate by set of previous definitions and by analogy with the Norden co-efficient "a", to set:

$$b = \frac{1}{2t_{od}^2} \tag{4.32}$$

Where t_{od} is the time at which the development curve exhibits a peak manning. This can then be used to obtain the following relation between the development time "t_d", and development peak manning, t_{od}:

$$t_{od} = \frac{t_d}{\sqrt{6}} \tag{4.33}$$

Relationship between "K_d" (total manpower cost of the development sub-cycle) and K (total manpower cost of the generic cycle) must be established. This can be obtained by using the observation that at the origin of time both cycles have the same slope. Thus from equation (4.21) and by differentiation of equation (4.29)

$$\left(\frac{dm}{dt}\right)_o = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2} = \left(\frac{dm_d}{dt}\right)_o$$

Consider equation (4.33), and we have

$$K_d = K/6 \tag{4.34}$$

It should also be noted that the difficulty D, is the same whether expressed in terms of "K" and "t_d" or "K_d" and "t_{od}". More formally, this can be stated by:

$$D = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2}$$

This does not apply to the manpower build up D_o

$$D_o = \frac{K}{t_d^3} = \frac{K_d}{\sqrt{6}\, t_{od}^3} \tag{4.35}$$

Note here that there is an extra factor $\sqrt{6}$ when the calculations are made at the sub cycle level.

Conte et. al., [CONT86] investigated the Putnam models and observed that they work reasonably well on very large systems, but seriously over estimate effort on medium or small size systems. The model emphasises heavily on the size and development schedule attributes while down playing with other attributes. The constant C must be used to reflect all other attributes including complexity, use of modern programming practices and personnel ability.

Example 4.15

A software development requires 90 PY during the total development sub-cycle. The development time is planned for a duration of 3 years and 5 months [CONT86]

(*a*) Calculate the manpower cost expended until development time.

(*b*) Determine the development peak time

(*c*) Calculate the difficulty and manpower build-up.

Solution

(*a*) Duration $t_d = 3.41$ years

We know from equation (4.31)

$$\frac{y_d(t_d)}{K_d} = 0.95$$

$$Y_d(t_d) = 0.95 \times 90$$

$$= 85.5 \text{ PY}$$

(*b*) We know from equation (4.33)

$$t_{od} = \frac{t_d}{\sqrt{6}} = 3.41/2.449 = 1.39 \text{ years}$$

$$\cong 17 \text{ months.}$$

(*c*) Total Manpower development

$$K_d = y_d(t_d)/0.95$$

$$= 85.5/0.95 = 90$$

$$K = 6K_d = 90 \times 6 = 540 \text{ PY}$$

$$D = K/t_d^2 = 540/(3.41)^2 = 46 \text{ person/years}$$

$$D_0 = \frac{K}{t_d^3} = 540/(3.41)^3 = 13.6 \text{ person/year}^2.$$

Example 4.16

A software development for avionics has consumed 32 PY upto development cycle and produced a size of 48000 LOC. The development of project was completed in 25 months. Calculate the development time, total manpower requirement, development peak time, difficulty, manpower build up and technology factor.

Solution

Development time t_d = 25 months = 2.08 years

Total manpower development $K_d = \dfrac{Y_d(t_d)}{0.95}$

$$K_d = \frac{32}{0.95} = 33.7 \text{ PY}$$

Development peak time $t_{od} = \dfrac{(t_d)}{\sqrt{6}}$

$$= 0.85 \text{ years (or 10 months).}$$

$$K = 6K_d = 6 \times 33.7 = 202 \text{ PY}$$

$$D = \frac{K}{t_d^2} = \frac{202}{(2.08)^2} = 46.7 \text{ person/year}$$

$$D_0 = \frac{K}{t_d^3} = \frac{202}{(2.08)^3} = 22.5 \text{ person/year}^2$$

Technology factor

$$C = SK^{-1/3} t_d^{-4/3}$$
$$= 48000 \times (202)^{-1/3} (2.08)^{-4/3}$$
$$= 3077.$$

Example 4.17

What amount of software can be delivered in 1 year 10 months in an organisation whose technology factor is 2400 if a total of 25 PY is permitted for development effort?

Solution

$$t_d = 1.8 \text{ years}$$
$$K_d = 25 \text{ PY}$$
$$K = 25 \times 6 = 150 \text{ PY}$$
$$C = 2400$$

We know $S = CK^{1/3} t_d^{4/3}$

$$= 2400 \times 5.313 \times 2.18 = 27920 \text{ LOC}$$

Example 4.18

The software development environment of an organisation developing real time software has been assessed at technology factor of 2200. The maximum value of manpower build up for this type of software is D_o = 7.5. The estimated size of the software to be developed is S = 55000 LOC [LOND87].

(*a*) Determine the total development time, the total development manpower cost, the difficulty and the development peak manning.

(*b*) The development time determined in (*a*) is considered too long. It is recommended that it be reduced by two months. What would happen?

Solution

We have $S = CK^{1/3}t_d^{4/3}$

$$\left(\frac{S}{C}\right)^3 = Kt_d^4$$

which is also equivalent to

$$\left(\frac{S}{C}\right)^3 = D_o t_d^7$$

then
$$t_d = \left[\frac{1}{D_0}\left(\frac{S}{C}\right)^3\right]^{1/7}$$

Since $\frac{S}{C} = 25,$

$t_d = 3$ years

As $K = D_0 t_d^3 = 7.5 \times 27 = 202$ PY

Total development manpower cost $K_d = \frac{202}{06} = 33.75$ PY

$$D = D_0 t_d = 22.5 \text{ person/year}$$

$$t_{od} = \frac{t_d}{\sqrt{6}} = \frac{3}{\sqrt{6}} = 1.2 \text{ years}$$

Using equations (4.20) and (4.29), we have

$$m_d(t) = 2K_d\, bte^{-bt^2}$$

$$Y_d(t) = K_d(1 - e^{-bt^2})$$

Here, $t = t_{od}$

Peak manning $= m_{od} = Dt_{od}e^{-1/2}$

$= 22.5 \times 1.2 \times .606 \cong 16$ persons

(*b*) Developing time reduction means either developing the software at a higher manpower build-up or producing less software.

(*i*) **Increase manpower build-up**

$$D_o = \frac{1}{t_d^7}\left(\frac{S}{C}\right)^3$$

The new development time would be 2.8 years and new manpower build-up is

$D_o = (25)^3/(2.8)^7 = 11.6$ person/year^2

$K = D_0 t^3{}_d = 254$ PY

$$K_d = \frac{254}{6} = 42.4 \text{ PY}$$

$$D = D_0 t_d = 32.5 \text{ person/year}$$

The peak time is t_{od} = 1.14 years

Peak manning $\quad m_{od} = Dt_{od}\, e^{-0.5}$

$$= 32.5 \times 1.14 \times 0.6 \cong 22 \text{ persons}$$

Note the huge increase in peak manning and manpower cost.

(*ii*) **Produce less software**

$$\left(\frac{S}{C}\right)^3 = D_0\, t_d^7 = 7.5 \times (2.8)^7 = 10119.696$$

$$\left(\frac{S}{C}\right)^3 = 21.62989$$

Then for $\quad C = 2200$

$$S = 47586 \text{ LOC}$$

The problem is now to decide which software functions can be cut down.

Example 4.19

A stand-alone project for which the size is estimated at 12500 LOC is to be developed in an environment such that the technology factor is 1200. Choosing a manpower build up $D_o = 15$, calculate the minimum development time, total development man power cost, the difficulty, the peak manning, the development peak time, and the development productivity.

Solution

Size (S) = 12500 LOC

Technology factor (C) = 1200

Manpower build up (D_o) = 15

Now $\quad S = CK^{1/3}t_d^{4/3}$

$$\frac{S}{C} = K^{1/3}t_d^{4/3}$$

$$\left(\frac{S}{C}\right)^3 = Kt_d^4$$

Also we know $\quad D_o = \dfrac{K}{t_d^3}$

$$K = D_o t_d^3 = D_o t_d^3$$

Hence $\quad \left(\dfrac{S}{C}\right)^3 = D_o t_d^7$

Substituting the values, we get

$$\left(\frac{12500}{1200}\right)^3 = 15t_d^7$$

$$t_d = \left[\frac{(10.416)^3}{15}\right]^{1/7}$$

$$t_d = 1.85 \text{ years}$$

(*i*) Hence Minimum development time (t_d) = 1.85 years.

(*ii*) Total development manpower cost $K_d = \frac{K}{6}$

Hence,
$$K = 15t_d^3$$
$$= 15(1.85)^3 = 94.97 \text{ PY}$$
$$K_d = \frac{K}{6} = \frac{94.97}{6} = 15.83 \text{ PY}$$

(*iii*) Difficulty $D = \frac{K}{t_d^2} = \frac{94.97}{(1.85)^2} = 27.75$ Person/year

(*iv*) Peak Manning $m_0 = \frac{K}{t_d\sqrt{e}}$
$$= \frac{94.97}{1.85 \times 1.648} = 31.15 \text{ Persons}$$

(*v*) Development Peak time $t_{od} = \frac{t_d}{\sqrt{6}}$
$$= \frac{1.85}{2.449} = 0.755 \text{ years}$$

(*vi*) Development Productivity
$$= \frac{\text{No. of lines of code (S)}}{\text{effort } (K_d)}$$
$$= \frac{12500}{15.83} = 789.6 \text{ LOC/PY.}$$

4.7 SOFTWARE RISK MANAGEMENT

We, software developers are extremely optimists. When planning software projects, we often assume that everything will go exactly as planned. Alternatively, we take the other extreme position. The creative nature of software development means we can never accurately predict what is going to happen, so what is the point of making detailed plans? Both these perspectives can lead to software surprises, when unexpected things happen that throw the project completely off track. Software surprises are never good news.

Risk Management is becoming recognised as an important area in the software industry to reduce this surprise factor. Risk management means dealing with a concern before it becomes a crisis. Therefore, most of the software development activities include risk management as a key part of the planning process and expect the plan to highlight the specific risk areas. The project planning is expected to quantify both probability of failure and consequences of failure and to describe what will be done to reduce the risk.

4.7.1 What is Risk?

Tomorrow's problems are today's risks. Hence, a simple definition of a "risk" is a problem that could cause some loss or threaten the success of the project, but which has not happened yet.

These potential problems might have an adverse impact on cost, schedule, or technical success of the project, the quality of our software products, or project team morale. Risk management is the process of identifying, addressing and eliminating these problems before they can damage the project.

We need to differentiate risks, as potential problems, from the current problems of the project. Different approaches are required to address these two kinds of issues. For example, a staff shortage because we have not been able to hire people with the right technical skills is a current problem; but the threat of our technical people being hired away by the competition is a risk. Current real problems require prompt, corrective action, whereas risk can be dealt with in several different ways. We might choose to avoid the risk entirely by changing the project approach or even cancelling the project.

Whether we tackle them head-on or keep our heads in the sand, risks have a potentially huge impact on many aspects of the project. We do far too much pretending in software. We pretend, we know who our users are, we know what their needs are, that we would not have staff turn over problems, that we can solve all technical problems that arise, that our estimates are achievable, and that nothing unexpected will happen.

Risk management is about discarding the rose-coloured glasses and confronting the very real potential of undesirable events conspiring to throw our project off track [WIEG98].

4.7.2 Typical Software Risks

The list of evil things that can befall a software project is depressingly long. Possible risks can come from group brainstorming activities, or from a risk factor chart accumulated from previous projects. There are no magic solutions to any of these risk factors, so we need to rely on past experience and a strong knowledge of contemporary software engineering and management practices to control these risks. Capers Jones has identified the top five risk factors that threaten projects in different applications [JONE94].

Dependencies

Many risks arise due to dependencies of project on outside agencies or factors. It is not easy to control these external dependencies. Some typical dependency-related risk factors are:

- Availability of trained, experienced people
- Intercomponent or inter-group dependencies
- Customer-furnished items or information
- Internal and external subcontractor relationships

Requirement issues

Many projects face uncertainty and turmoil around the product's requirements. While some of this uncertainty is tolerable in early stages, but the threat to success increases if such issues are not resolved as the project progresses. If we do not control requirements-related risk factors, we might either build the wrong product, or build the right product badly. Either situation results in unpleasant surprises and unhappy customers. Some typical factors are:

- Lack of clear product vision
- Lack of agreement on product requirements

- Unprioritized requirements
- New market with uncertain needs
- Rapidly changing requirements
- Inadequate impact analysis of requirements changes

Management issues

Project Managers usually write the risk management plan, and most people do not wish to air their weaknesses (assuming they even recognise them) in public. Nonetheless, issues like those listed below can make it harder for projects to succeed. If we do not confront such touchy issues, we should not be surprised if they bite us at some point. Defined project tracking processes, and clear roles and responsibilities, can address some of these risk factors.

- Inadequate planning and task identification
- Inadequate visibility into actual project status
- Unclear project ownership and decision making
- Unrealistic commitments made, sometimes for the wrong reasons
- Managers or customers with unrealistic expectations
- Staff personality conflicts
- Poor communication

Lack of knowledge

The rapid rate of change of technologies, and the increasing change of skilled staff, mean that our project teams may not have the skills we need to be successful. The key is to recognise the risk areas early enough so that we can take appropriate preventive actions, such as obtaining training, hiring consultants, and bringing the right people together on the project team. Some of the factors are:

- Inadequate training
- Poor understanding of methods, tools, and techniques
- Inadequate application domain experience
- New technologies
- Ineffective, poorly documented, or neglected processes

Other risk categories

The list of potential risk areas is long. Some of the critical areas are:

- Unavailability of adequate testing facilities
- Turnover of essential personnel
- Unachievable performance requirements
- Technical approaches that may not work

4.7.3 Risk Management Activities

Risk management involves several important steps, each of which is illustrated in Fig. 4.9. We should assess the risks on the project, so that we understand what may occur during the

course of development or maintenance. The assessment consists of three activities: identifying the risks, analysing them, and assigning priorities to each of them.

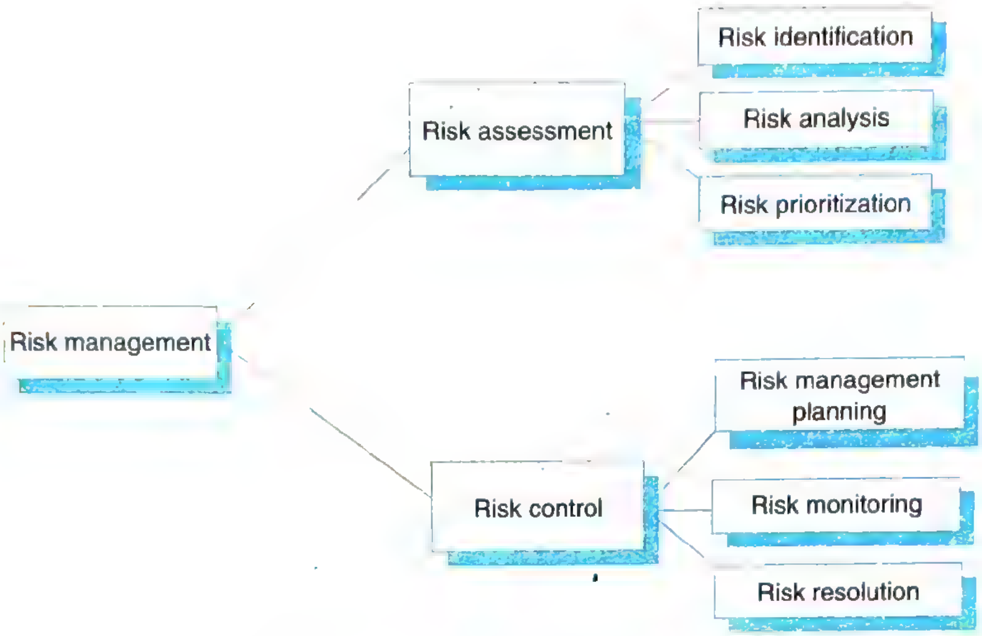


Fig. 4.9: Risk management activities

Risk assessment

It is the process of examining a project and identifying areas of potential risk. Risk identification can be facilitated with the help of a checklist of common risk areas of software projects; or by examining the contents of an organizational database of previously identified risks. Risk analysis involves examining how project outcomes might change with modification of risk input variables. Risk prioritization helps the project focus on its most severe risks by assessing the risk exposure. Exposure is the product of the probability of incurring a loss due to the risk and the potential magnitude of that loss. This prioritization can be done in a quantitative way, by estimating the probability (0.1 – 1.0) and relative loss, on a scale of 1 to 10. Multiplying these factors together provide an estimation of risk exposure due to each risk item, which can run from 0.1 (do not give it another thought) through 10 (stand back, here it comes!). The higher the exposure, the more aggressively the risk should be tackled. It may be easier to simply estimate both probability and impact as High, Medium, or Low. Those items having at least one dimension rated as High are the ones to worry about first.

Another way of handling risk is the risk avoidance. Do not do the risky things! We may avoid risks by not undertaking certain projects, or by relying on proven rather than cutting edge technologies.

Risk control

It is the process of managing risks to achieve the desired outcomes. Risk management planning produces a plan for dealing with each significant risk. It is useful to record decisions in the plan, so that both customer and developer can review how problems are to be avoided, as well as how they are to be handled when they arise. We should also monitor the project as

development progresses, periodically revaluating the risks, their probability, and likely impact. Risk resolution is the execution of the plans for dealing with each risk.

Simply identifying the risks of any project is not enough. We should write them down in a way that communicates the nature and status of risks over the duration of the project.

REFERENCES

[BASL80] Basil V.R., "*Resource Models: Models & Metrics for Software Management and Engineering*", IEEE, pp-4–9, 1980.

[BOEH81] Boehm B.W., "*Software Engineering Economics*", Prentice -Hall, 1981.

[UCSD01] "COCOMO-II" *Model Definition Manual*", Version 1.4, University of Southern California, 2001.

[GHEZ94] Ghezzi C., et Al., "*Software Engineering*", PHI, 1994.

[HUMP95] Humphrey Watts S., "*A Discipline for Software Engineering*", Addison-Wesley, Pub. Co., 1995.

[JONE91] Jones C, "*Applied Software Measurement, Assuring Productivity & Quality*", McGraw Hill, New York, NY, 1991.

[JONE94] Jones C., "*Assessment and Control of Software Risks*", Englewood Cliffs, N.J., Prentice-Hall, 1994.

[LOND87] Londeix B., "*Cost Estimation for Software Development*", Addison -Wesley Pub. Co., 1987.

[NORD58] Norden P.V., "*Curve Fitting for a Model of Applied Research and Development Scheduling*", IBM Journal, Research & Development, Vol 3, No.2, PP.232–248, July, 1958.

[NORD77] Norden P.V., "*Project Life Cycle Modeling: Background and Application of the Life Cycle Curves*", US Army Computer Systems Command, 1977.

[PRESS2K] Pressman Roger, "*Software Engineering*", McGraw Hill Pub., 2000.

[PUTN84] Putnam I.H., Putnam D.T., "*A Verification of the Software Fourth Power Trade off Law*", Proc. Of the Int. Soc. of para-metric Analysis 3, 1, May 184, 443–471

[SAGE90] Sage A.P., & Palmer J.D., "*Software System Engineering*", John Wiley & Sons Pub. Co., 1990.

[SOMM96] Summerville Ian, "*Software Engineering*", Addison - Wesley Pub Co., 1996.

[WALS77] Walson C.E. & Felix C.P., "*A Method for Programming Measurement and Estimation*", IBM System Journal, 16(1), pp - 54–73, 1977.

[WIEG98] Wiegers K.E., "*Know Your Enemy: Software Risk Management*", Software Development Magazine, October, 1998.

MULTIPLE CHOICE QUESTIONS

Note: *Choose most appropriate answer of the following questions*

4.1. After the finalisation of SRS, we may like to estimate

(*a*) size (*b*) cost

(*c*) development time (*d*) all of the above.

4.2. Which one is not a size measure for software

(*a*) LOC (*b*) function Count

(*c*) cyclomatic Complexity (*d*) halstead's program length.

4.3. Function count method was developed by

(*a*) B. Beizer (*b*) B. Boehm

(*c*) M. Halstead (*d*) Alan Albrecht.

4.4. Function point analysis (FPA) method decomposes the system into functional units. The total number of functional units are

(*a*) 2 (*b*) 5

(*c*) 4 (*d*) 1.

4.5. IFPUG Stand for

(*a*) initial function point uniform group (*b*) international function point uniform group

(*c*) international function point user group (*d*) initial function point user group.

4.6. Function point can be calculated by

(*a*) UFP * CAF (*b*) UFP * FAC

(*c*) UFP * Cost (*d*) UFP * Productivity.

4.7. Putnam resource allocation model is based on

(*a*) function points

(*b*) Norden/Rayleigh curve

(*c*) Putnam theory of software management

(*d*) Boehm's observations on manpower utilisation rate.

4.8. Manpower buildup for Putnam resource allocation model is

(*a*) K/t_d^2 persons/year^2 (*b*) K/t_d^3 persons/year^2

(*c*) K/t_d^2 persons/year (*d*) K/t_d^3 persons/year.

4.9. COCOMO was developed initially by

(*a*) B.W. Bohem (*b*) Gregg Rothermal

(*c*) B. Beizer (*d*) Rajeev Gupta.

4.10. A COCOMO model is

(*a*) common cost estimation model (*b*) constructive cost estimation model

(*c*) complete cost estimation model (*d*) comprehensive cost estimation model.

4.11. Estimation of software development effort for organic software in COCOMO is

(*a*) $E = 2.4(KLOC)^{1.05}$ PM (*b*) $E = 3.4(KLOC)^{1.06}$ PM

(*c*) $E = 2.0(KLOC)^{1.05}$ PM (*d*) $E = 2.4(KLOC)^{1.07}$ PM.

4.12. Estimation of size for a project is dependent on

(*a*) cost (*b*) schedule

(*c*) time (*d*) none of the above.

4.13. In function point analysis, number of complexity adjustment factors are

(*a*) 10 (*b*) 20

(*c*) 14 (*d*) 12.

4.14. COCOMO-II estimation model is based on

(*a*) complex approach (*b*) algorithmic approach

(*c*) bottom up approach (*d*) top down approach.

4.15. Cost estimation for a project may include

(*a*) software cost (*b*) hardware cost

(*c*) personnel costs (*d*) all of the above.

4.16. In COCOMO model, if project size is typically 2 – 50 KLOC, then which mode is to be selected?
(a) organic
(b) semidetached
(c) embedded
(d) none of the above.

4.17. COCOMO-II was developed at
(a) university of Maryland
(b) university of Southern California
(c) IBM
(d) AT & T Bell labs

4.18. Which one is not a Category of COCOMO-II?
(a) end user programming
(b) infrastructure sector
(c) requirement sector
(d) system Integration.

4.19. Which one is not an infrastructure software?
(a) operating system
(b) database management system
(c) compilers
(d) result management system.

4.20. How many stages are in COCOMO-II?
(a) 2
(b) 3
(c) 4
(d) 5.

4.21. Which one is not a stage of COCOMO-II?
(a) application composition estimation model
(b) early design estimation model
(c) post architecture estimation model
(d) comprehensive cost estimation model.

4.22. In Putnam resource allocation model, Rayleigh curve is modeled by the equation
(a) $m(t) = 2at\ e^{-at^2}$
(b) $m(t) = 2Kt\ e^{-at^2}$
(c) $m(t) = 2Kat\ e^{-at^2}$
(d) $m(t) = 2Kbt\ e^{-at^2}$.

4.23. In Putnam resource allocation model, technology factor 'C' is defined as
(a) $C = SK^{-1/3}t_d^{-4/3}$
(b) $C = SK^{1/3}t_d^{4/3}$.
(c) $C = SK^{1/3}t_d^{-4/3}$
(d) $C = SK^{-1/3}t_d^{4/3}$.

4.24. Risk management activities are divided in
(a) 3 categories
(b) 2 categories
(c) 5 categories
(d) 10 categories.

4.25. Which one is not a risk management activity?
(a) risk assessment
(b) risk control
(c) risk generation
(d) none of the above.

EXERCISE

4.1. What are various activities during software project planning?

4.2. Describe any two software size estimation techniques.

4.3. A proposal is made to count the size of 'C' programs by number of semicolons, except those occurring with literal strings. Discuss the strengths and weaknesses to this size measure when compared with the lines of code count.

4.4. Design a LOC counter for counting LOC automatically. Is it language dependent? What are the limitations of such a counter?

4.5. Compute the function point value for a project with the following information domain characteristics.

Number of user inputs = 30

Number of user outputs = 42

Number of user enquiries = 08

Number of files = 07

Number of external interfaces = 6

Assume that all complexity adjustment values are moderate.

4.6. Explain the concept of function points. Why FPs are becoming acceptable in industry?

4.7. What are size metrics? How is function point metric advantageous over LOC metric? Explain.

4.8. Is it possible to estimate software size before coding? Justify your answer with suitable examples.

4.9. Describe the Albrecht's function count method with a suitable example.

4.10. Compute the function point FP for a payroll program that reads a file of employees and a file of information for the current month and prints cheques for all the employees. The program is capable of handling an interactive command to print an individually requested cheque immediately.

4.11. Assume that the previous payroll program is expected to read a file containing information about all the cheques that have been printed. The file is supposed to be printed and also used by the program next time it is run, to produce a report that compares payroll expenses of the current month with those of the previous month. Compute function points for this program. Justify the difference between the function points of this program and previous one by considering how the complexity of the program is affected by adding the requirement of interfacing with another application (in this case, itself).

4.12. Explain the Walson & Felix model and compare with the SEL model.

4.13. The size of a software product to be developed has been estimated to be 22000 LOC. Predict the manpower cost (effort) by Walston-Felix Model and SEL Model.

4.14. A database system is to be developed. The effort has been estimated to be 100 Persons-Months. Calculate the number of lines of code and productivity in LOC/Person-Month.

4.15. Discuss various types of COCOMO mode. Explain the phase wise distribution of effort.

4.16. Explain all the levels of COCOMO model. Assume that the size of an organic software product has been estimated to be 32,000 lines of code. Determine the effort required to develop the software product and the nominal development time.

4.17. Using the basic COCOMO model, under all three operating modes, determine the performance relation for the ratio of delivered source code lines per person-month of effort. Determine the reasonableness of this relation for several types of software projects.

4.18. The effort distribution for a 240 KLOC organic mode software development project is: product design 12%, detailed design 24%, code and unit test 36%, integrate and test 28%. How would the following changes, from low to high, affect the phase distribution of effort and the total effort: analyst capability, use of modern programming languages, required reliability, requirements volatility?

4.19. Specify, design, and develop a program that implements COCOMO. Using reference [BOEH81] as a guide, extend the program so that it can be used as a planning tool.

4.20. Suppose a system for office automation is to be designed. It is clear from requirements that there will be five modules of size 0.5 KLOC, 1.5 KLOC, 2.0 KLOC, 1.0 KLOC and 2.0 KLOC respectively. Complexity, and reliability requirements are high. Programmer's capability and experience is

low. All other factors are of nominal rating. Use COCOMO model to determine overall cost and schedule estimates. Also calculate the cost and schedule estimates for different phases.

4.21. Suppose that a project was estimated to be 600 KLOC. Calculate the effort and development time for each of the three modes *i.e.*, organic, semidetached and embedded.

4.22. Explain the COCOMO-II in detail. What types of categories of projects are identified?

4.23. Discuss the Infrastructure Sector of COCOMO-II.

4.24. Describe various stages of COCOMO-II. Which stage is more popular and why?

4.25. A software project of application generator category with estimated size of 100 KLOC has to be developed. The scale factor (B) has high precedentness, high development flexibility. Other factors are nominal. The cost drivers are high reliability, medium database size, high Personnel capability, high analyst capability. The other cost drivers are nominal. Calculate the effort in Person-months for the development of the project.

4.26. Explain the Putnam resource allocation model. What are the limitations of this model?

4.27. Describe the trade-off between time versus cost in Putnam resource allocation model.

4.28. Discuss the Putnam resource allocation model. Derive the time and effort equations.

4.29. Assuming the Putnam model, with $S = 100{,}000$, $C = 5000$, $D_o = 15$, Compute development time t_d and manpower development K_d.

4.30. Obtain software productivity data for two or three software development programs. Use several cost estimating models discussed in this chapter. How do the results compare with actual project results?

4.31. It seems odd that cost and size estimates are developed during software project planning—before detailed software requirements analysis or design has been conducted. Why do we think this is done? Are there circumstances when it should not be done?

4.32. Discuss typical software risks. How staff turnover problem affects software projects?

4.33. What are risk management activities? Is it possible to prioritize the risk?

4.34. What is risk exposure? What techniques can be used to control each risk?

4.35. What is risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?

4.36. There are significant risks even in student projects. Analyse a student project and list the risks.

Software Design

5

Software design is more creative process than analysis because it deals with the development of the actual mechanics for a new workable system. While analysing, it is possible to produce the correct model of an existing system. However, there is, no such thing as correct design. Good design is always system dependent and what is good design for one system may be bad for another.

The design of the new system must be done in great detail as it will be the basis for future computer programming and system implementation. Design is a problem-solving activity and as such, very much a matter of trial and error. The designer, together with users, has to search for a solution using his/her professional wisdom until there is an agreement that a satisfactory solution has been found.

For small projects (such as student's projects), one can sit with the specifications and simply write a program. For larger projects, it is necessary to bridge the gap between specifications and the coding with something more concrete. This bridge is the software design.

5.1 WHAT IS DESIGN?

Design is the highly significant phase in the software development where the designer plans "how" a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain. A software requirements specifications (SRS) document tells us "what" a system does, and becomes input to the design process, which tells us "how" a software system works. Designing software systems means determining how requirements are realized and result is a software design document (SDD). Thus, the purpose of design phase is to produce a solution to a problem given in SRS document.

A framework of the design is given in Fig. 5.1. It starts with initial requirements and ends up with the final design. Here, data is gathered on user requirements and analysed accordingly. A high level design is prepared after answering questions of requirements. Moreover, design is validated against requirements on regular basis. Design is refined in every cycle and finally it is documented to produce software design document. Fig. 5.1 shows the design framework.

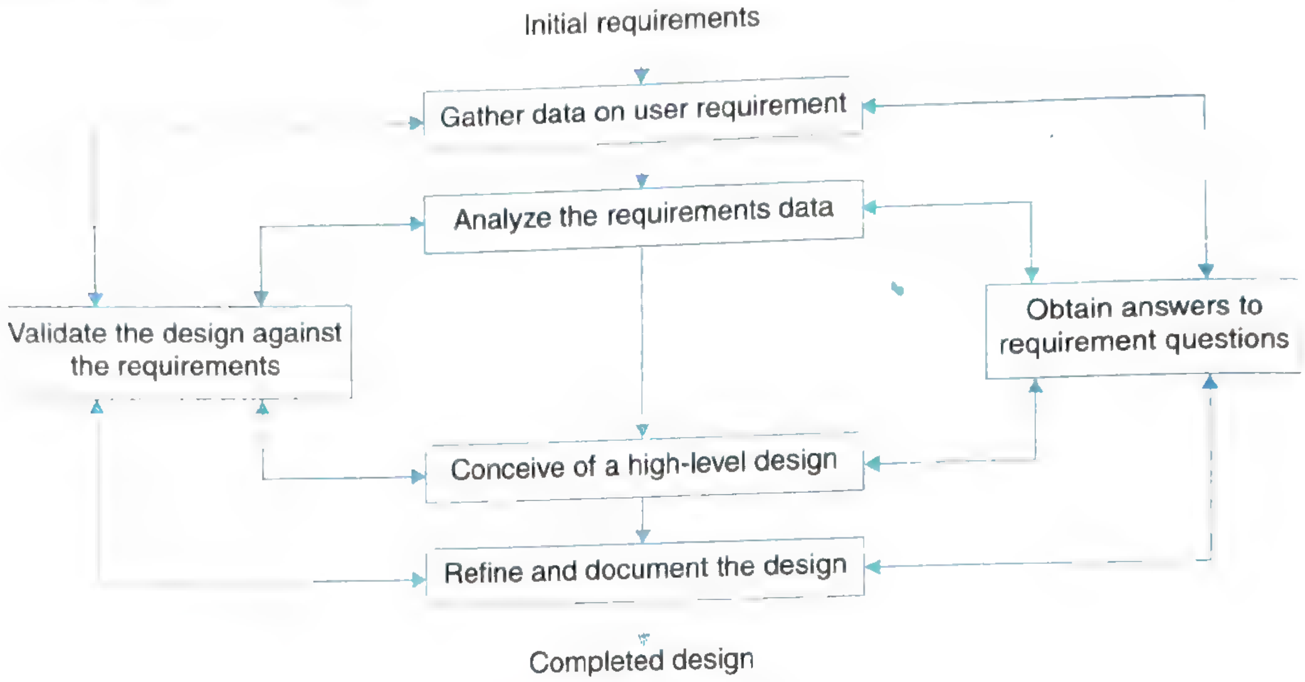


Fig. 5.1: Design framework

5.1.1 Conceptual and Technical Designs

The process of software design involves the transformation of ideas into detailed implementation descriptions, with the goal of satisfying the software requirements. To transform requirements into a working system, designers must satisfy both customers and the system builders (coding persons). The customers understand what the system is to do. At the same time, the system builders must understand how the system is to work. For this reason, design is really a two part, iterative process. First, we produce conceptual design that tells the customer exactly what the system will do. Once the customer approves the conceptual design, we translate the conceptual design into a much more detailed document, the technical design, that allows system builders to understand the actual hardware and software needed to solve the customer's problem. This two part design process is shown in Fig. 5.2.

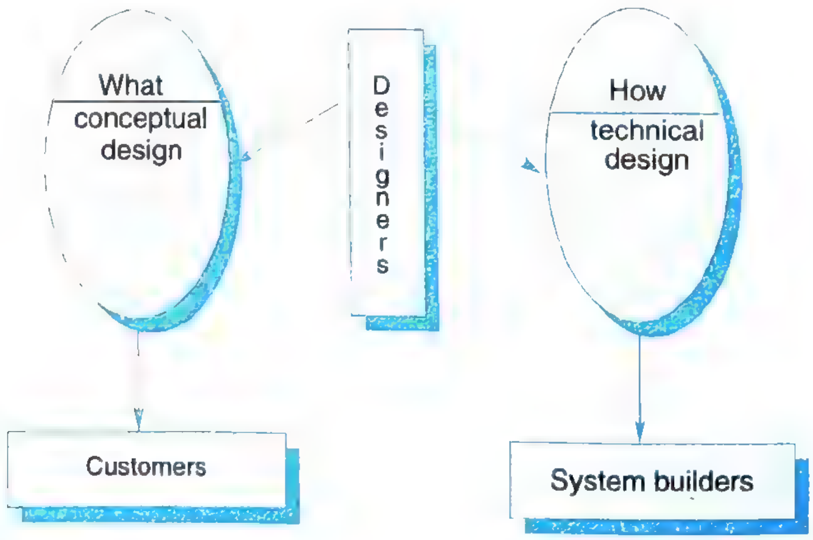


Fig. 5.2: A two-part design process

The two design documents describe the same system, but in different ways because of the different audiences for the documents. The conceptual design answers the following questions [PFLE98].

- Where will the data come from?
- What will happen to the data in the system?
- How will the system look to users?
- What choices will be offered to users?
- What is the timing of events?
- How will the reports and screens look like?

The conceptual design describes the system in language understandable to the customer. It does not contain any technical jargons and is independent of implementation.

By contrast, the technical design describes the hardware configuration, the software needs, the communications interfaces, the input and output of the system, the network architecture, and anything else that translates the requirements into a solution to the customer's problem.

Sometimes customers are very sophisticated and they can understand the "what" and "how" together. This can happen when customers are themselves software developers and may not require conceptual design. In such a cases comprehensive design document may be produced.

5.1.2 Objectives of Design

The specification (*i.e.* the "outside" view) of a program should obviously be as free as possible of aspects imposed by "how" the program will work (*i.e.* the "inside" view). It is seldom a document from which coding can directly be done. So design fills the gap between specifications and coding; taking the specifications, deciding how the program will be organized, and the methods it will use, in sufficient detail as to be directly codeable.

If the specification calls for a large or complex program (or both), then the design is quite likely to work down through a number of levels. At each level, breaking the implementation problem into a combination of smaller and simpler problems. Filling a large gap will involve a number of stepping-stones! The wider the gap, the larger the number of stepping-stones. The design needs to be

- Correct and complete
- Understandable
- At the right level
- Maintainable, and to facilitate maintenance of the produced code.

Software designers do not arrive at a finished design document immediately but develop the design iteratively through a number of different phases. The design process involves adding details as the design is developed with constant backtracking to correct earlier, less formal, designs. The starting point is an informal design which is refined by adding information to make it consistent and complete and this is shown in Fig. 5.3 [SOMM2K].

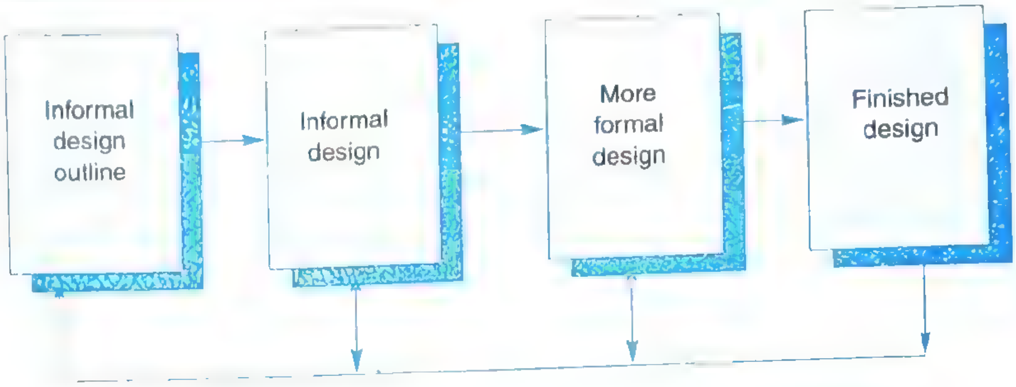


Fig. 5.3: The transformation of an informal design to a detailed design

5.1.3 Why Design is Important?

A good design is the key to successful product. Almost 2000 years ago Roman Architect Vitruvious recorded the following attributes of a good design:

- Durability
- Utility and
- Charm

A well-designed system is easy to implement, understandable and reliable and allows for smooth evolution. Without design, we risk building an unstable system:

- One that will fail when small changes are made
- One that will be difficult to maintain
- One whose quality cannot be assessed until late in the software process.

Therefore, software design should contain a sufficiently complete, accurate and precise solution to a problem in order to ensure its quality implementation.

There are three characteristics that serve as a guide for the evolution of a good design.

- The design must implement all of the explicit requirements contained in the analysis model and it must accommodate all of the implicit requirements desired by the customer.
- The design must be readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional and behavioural domain from an implementation perspective.

5.2 MODULARITY

There are many definitions of the term "module". They range from "a module is a FORTRAN subroutine" to "a module is an Ada package" to "procedures and functions of PASCAL and C", to "C++ / Java Classes", to "Java packages" to "a module is a work assignment for an individual programmer" [FAIR2K]. All of these definitions are correct. A modular system consist of well defined, manageable units with well defined interfaces among the units. Desirable properties of a modular system include:

- Each module is a well defined subsystem that is potentially useful in other applications
- Each module has a single, well defined purpose
- Modules can be separately compiled and stored in a library
- Modules can use other modules
- Modules should be easier to use than to build
- Modules should be simpler from the outside than from the inside

Modularity is the single attribute of software that allows a program to be intellectually manageable [MYER78]. It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

A system is considered modular if it consists of discreet components so that each component can be implemented separately and a change to one component has minimal impact on other components. Here, one important question arises is to what extent we shall modularize. As the number of modules grows, the effort associated with integrating the module also grows. Fig. 5.4 establishes the relationship between cost/effort and number of modules in a software.

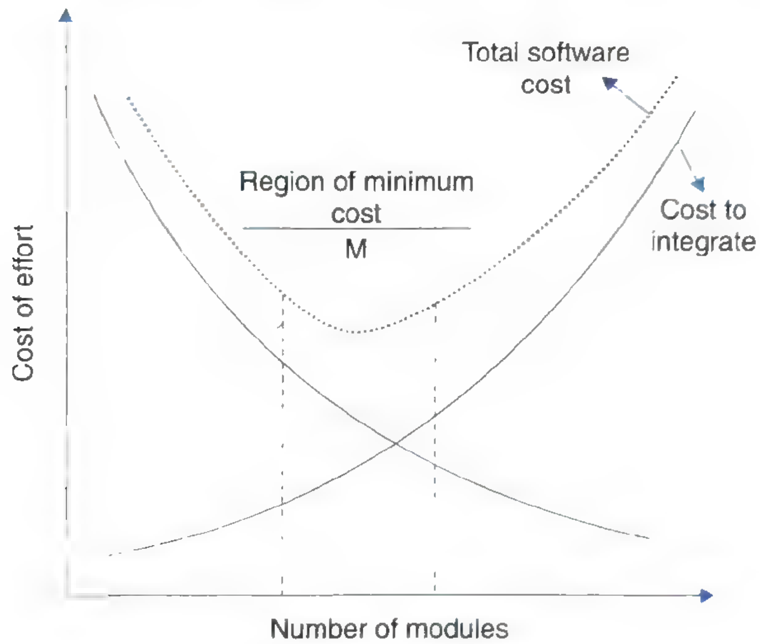


Fig. 5.4: Modularity and software cost

It can be observed that a software system cannot be made modular by simply chopping it into a set of modules. Each module needs to support a well defined abstraction and should have a clear interface through which it can interact with other modules. Thus, it is felt that under modularity and over modularity in a software should be avoided.

5.2.1 Module Coupling

Coupling is the measure of the degree of interdependence between modules. Two modules with high coupling are strongly interconnected and thus, dependent on each other. Two modules with low coupling are not dependent on one another. "Loosely coupled" systems are made up of

modules which are relatively independent. "Highly coupled" systems share a great deal of dependence between modules. For example, if modules make use of shared global variables. 'Uncoupled' modules have no interconnections at all; they are completely independent as shown in Fig. 5.5.

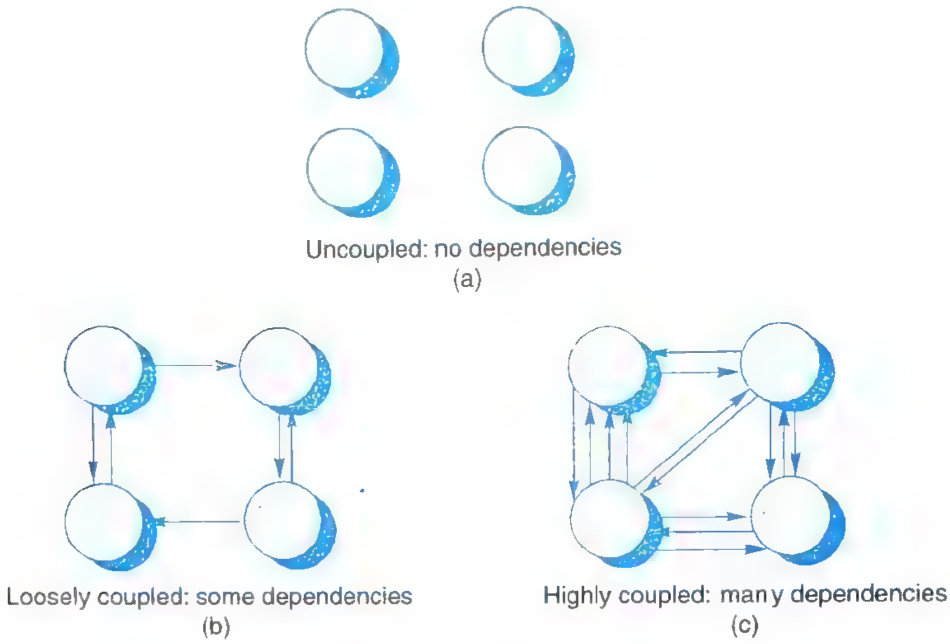


Fig. 5.5: Module coupling

A good design will have low coupling. Thus, interfaces should be carefully specified in order to keep low value of coupling.

Coupling is measured by the number of interconnections between modules. For example, coupling increases as the number of calls between modules increases, or the amount of shared data increases. The hypothesis is that design with high coupling will have more errors. Loose coupling, on the other hand, minimizes the interdependence amongst modules. This can be achieved in the following ways:

- Controlling the number of parameters passed amongst modules
- Avoid passing undesired data to calling module
- Maintain parent/child relationship between calling and called modules
- Pass data, not the control information

Fig. 5.6 demonstrates two alternative design for editing a student record in a "Student Information System".

The first design demonstrates tight coupling wherein unnecessary information as student name, student address, course is passed to the calling module. Passing superfluous information unnecessary increases the overhead, reducing the system performance/efficiency.

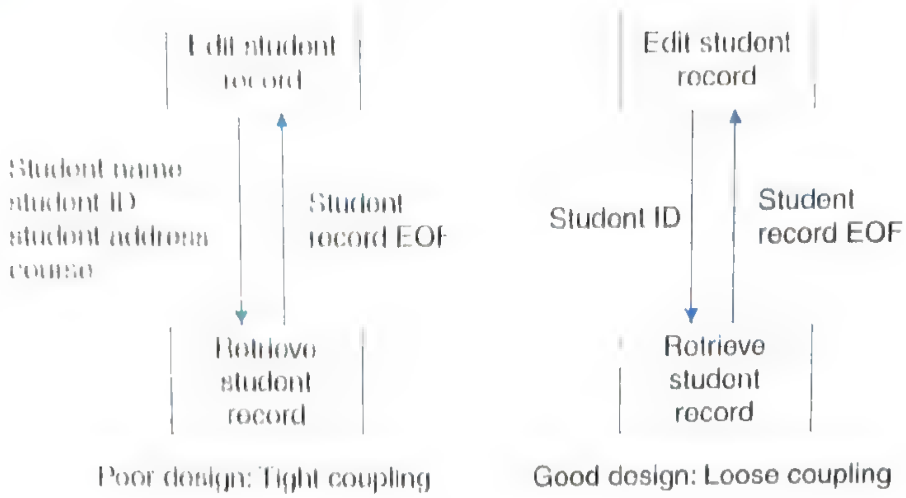


Fig. 5.6: Example of coupling

Types of coupling

Different types of coupling are content, common, external, control, stamp and data. The strength of coupling from lowest coupling (best) to highest coupling (worst) is given in Fig. 5.7.

Data coupling	Best
Stamp coupling	↑
Control coupling	
External coupling	
Common coupling	
Content coupling	(Worst)

Fig. 5.7: The types of module coupling

Given two procedures A and B, we can identify a number of ways in which they can be coupled.

Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent. A good strategy is to ensure that no module communication contains "tramp data". In Fig. 5.6 above students name, address, course are examples of tramp data that are unnecessarily communicated between modules. By ensuring that modules communicate only necessary data, module dependency is minimized.

Stamp coupling

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another. Since not all data making up the structure are usually necessary in

communication between the modules, stamp coupling typically involves tramp data. If one procedure only needs a part of a data structure, calling module should pass just that part, not the complete data structure.

Control coupling

Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

External coupling

A form of coupling in which a module has a dependency to other module, external to the software being developed or to a particular type of hardware. This is basically related to the communication to external tools and devices.

Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of change. With common coupling, it can be difficult to determine which module is responsible for having set a variable to a particular value. Fig. 5.8 shows how common coupling works [PFLE98]

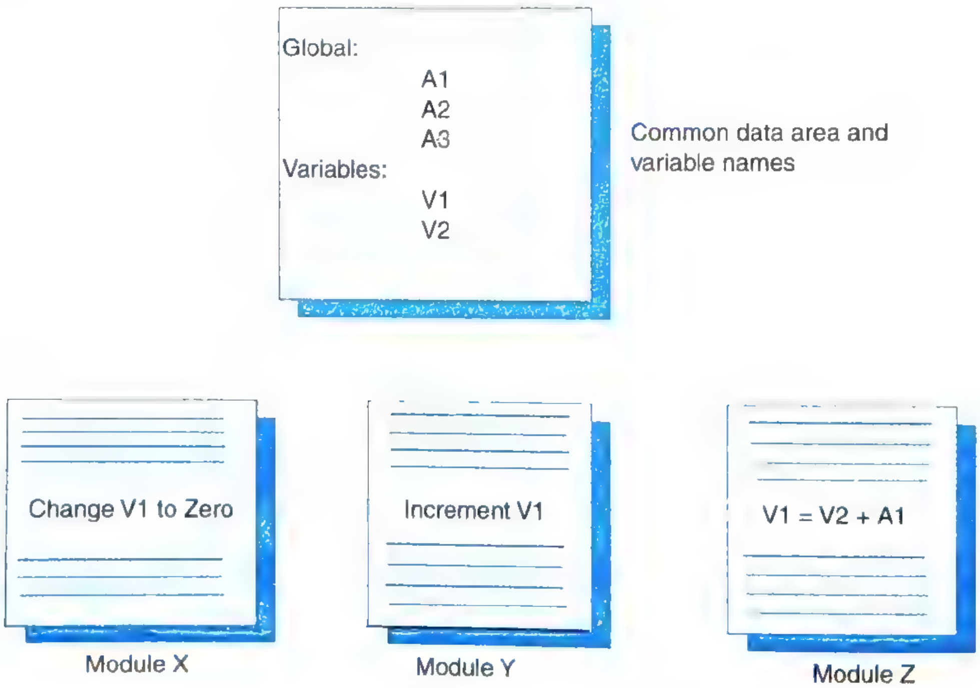


Fig. 5.8: Example of common coupling

Content coupling

Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. In Fig. 5.9, module B branches into D, even though D is supposed to be under the control of C.

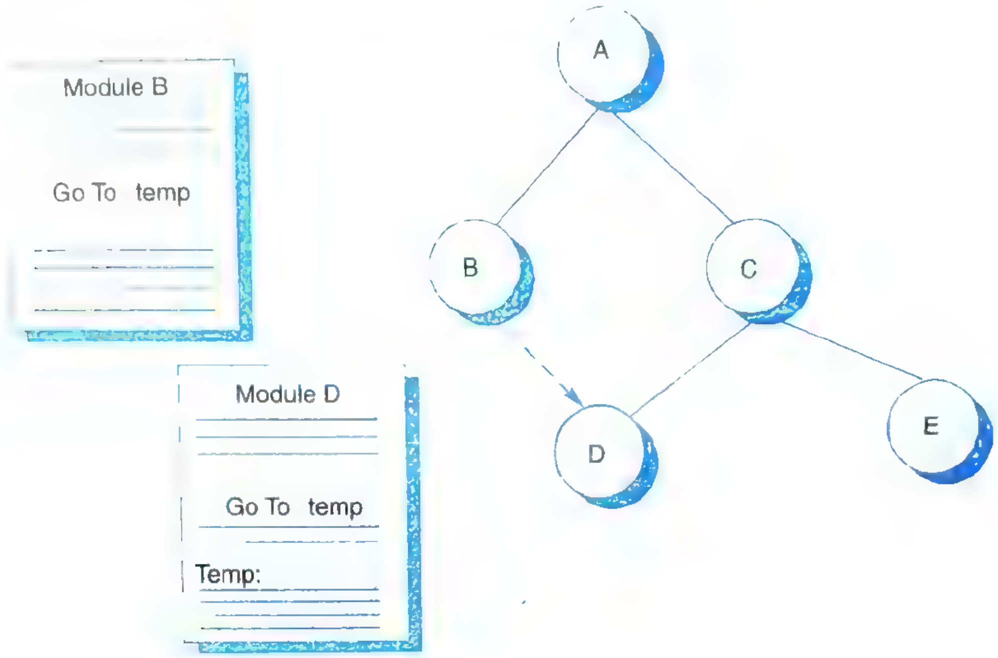


Fig. 5.9: Example of content coupling

5.2.2 Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related. A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with other modules. This is shown in Fig. 5.10. Cohesion may be viewed as a glue that keeps the module together. It is a measure of the mutual officity of the components of a module.

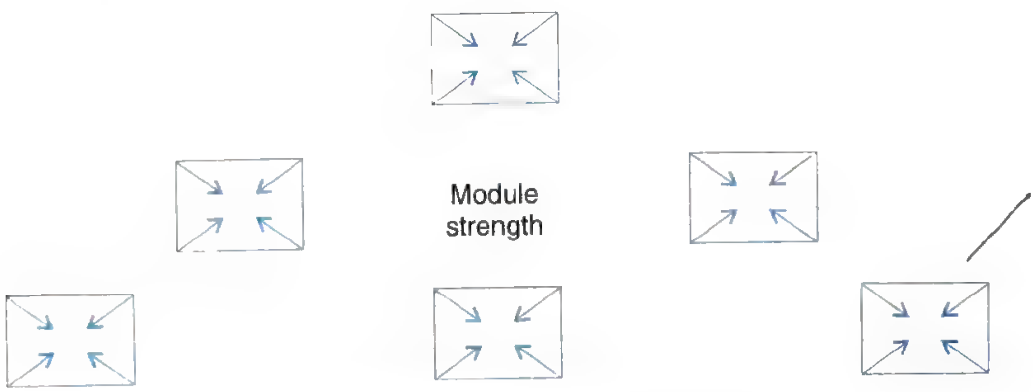


Fig. 5.10: Cohesion = Strength of relations within modules

Thus, we want to maximize the interaction within a module. Hence, an important design objective is to maximize the module cohesion and minimize the module coupling.

Types of cohesion

There are seven types or levels of cohesion and are shown in Fig. 5.11. Given a procedure that carries out operations X and Y, we can describe various forms of cohesion between X and Y.

Functional cohesion	Best (high)
Sequential cohesion	↑
Communicational cohesion	│
Procedural cohesion	│
Temporal cohesion	│
Logical cohesion	│
Coincidental cohesion	Worst (low)

Fig. 5.11: Types of module cohesion

Functional cohesion

X and Y are part of a single functional task. This is very good reason for them to be contained in the same procedure. Such a module often transformed a single input datum into a single output datum. The mathematical subroutines such as 'calculate current GPA' or 'cumulative GPA' are typical examples of functional cohesion.

Sequential cohesion

X outputs some data which forms the input to Y. This is the reason for them to be contained in the same procedure.

For example, addition of marks of individual subjects into a specific format is used to calculate the GPA as input for preparing the result of the students.

A component is made of parts that need to communicate/exchange data form one source for different functional purposes. They are together in a component for communicational convenience. For example calculate current and cumulative GPA uses the "student grade record" as input.

Communicational cohesion

X and Y both operate on the same input data or contribute towards the same output data. This is okay, but we might consider making them separate procedures.

Procedural cohesion

X and Y are both structured in the same way. This is a poor reason for putting them in the same procedure. Thus, procedural cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed. These types of modules are typically the result of first flow charting the solution to a program and then selecting a sequence of instructions to serve as a module. Since these modules consist of instructions that accomplish several tasks that are virtually unrelated these types of modules tend to be less maintainable. For example, if a report module of an examination system includes the following "calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA" is a case of procedural cohesion.

Temporal cohesion

X and Y both must perform around the same time. So, module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span. The set of functions responsible for initialization, start up activities such as setting program counters or control flags associated with programs exhibit temporal cohesion. This is not a good reason to put them in same procedure.

Logical cohesion

X and Y perform logically similar operations. Therefore, logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions. Considerable duplication can exist in the logical strength level. For example, more than one data item in an input transaction may be a date. Separate code would be written to check that each such date is a valid date. A better way to construct a DATECHECK module and call this module whenever a date check is necessary.

Coincidental cohesion

X and Y here no conceptual relationship other than shared code. Hence, coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another. That is, instead of creating two components, each of one part, only one component is made with two unrelated parts. For example, check validity and print is a single component with two parts. Coincidental cohesion is to be avoided as far as possible.

5.2.3 Relationship between Cohesion and Coupling

The essence of the design process is that the system is decomposed into parts to facilitate the capability of understanding and modifying a system. Projects rarely gets into trouble because of massive requirement changes. These changes can be properly recognized and properly reviewed.

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a good software design professes clean decomposition of a problem into modules and the arrangement of these modules in a neat hierarchy. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

A good example of a system that has high cohesion and low coupling is the 'plug and play' feature of the computer system. Various slots in the mother board of the system simply facilitate to add or remove the various services/functionalities without affecting the entire system. This is because the add on components provide the services in highly cohesive manner. Fig. 5.12 provides a graphical review of cohesion and coupling.

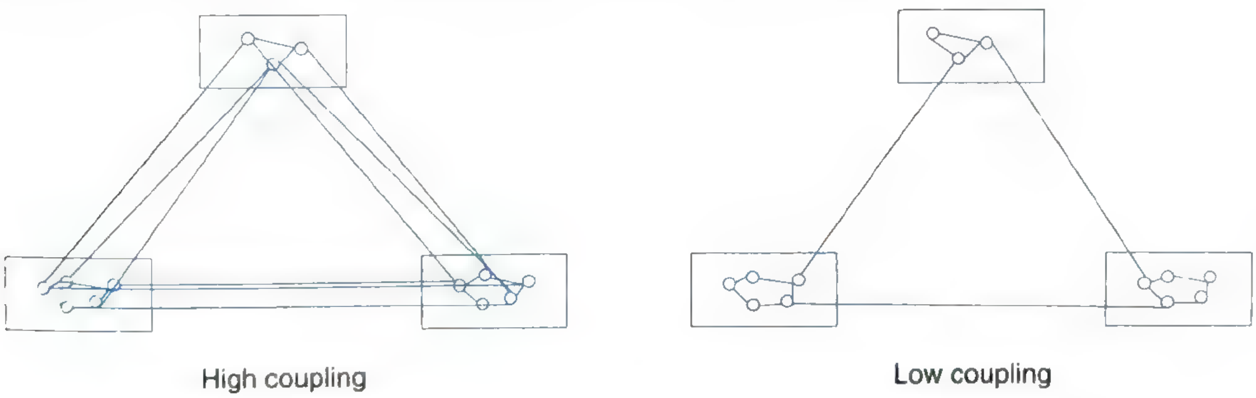


Fig. 5.12: View of cohesion and coupling

Module design with high cohesion and low coupling characterizes a module as black box when the entire structure of the system is described. Each module can be dealt separately when the module functionality is described.

5.3 STRATEGY OF DESIGN

A good system design strategy is to organize the program modules in such a way that are easy to develop and later to, change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:

- First, even pre-existing code, if any, needs to be understood, organized and pieced together.
- Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system.

In early days, if any design was done, it was just "writing down the flowchart in words". Many people feel that flowcharts are too detailed, so leading to the detail often being decided too early, and too far from the specifications. Hence, there is a sudden jump from specifications to flow chart that leads to the cause of many errors. Flowcharts are at a low level. As a result, errors in flow-charts could only be found by coding them, seeing that the code ran wrongly, diagnosing that the error is in the flow chart, then diagnosing where in the flowchart, then fixing it, modifying code and recording. Repeated surgery on the flow chart sometimes lead to the final flow chart where further errors can not be fixed and the project may fail.

So writers of large and complex software now seldom use flow charts for design. The result is that we have designed other notations for expressing designs, and they are at a "higher level" than flow charts. This helps us to minimize the length of jumps from specifications to design and design to code. These notations usually permit multiple levels of design, and many small jumps in place of one or two massive jumps.

There are many strategies or techniques for performing system design. They include bottom up approach, top down approach, and hybrid approach.

5.3.1 Bottom-Up Design

A common approach is to identify modules that are required by many programs. These modules are collected together in the form of a "library". These modules may be for match functions, for input-output functions, for graphical functions etc. We may have collections of modules for result preparation system like "maintain student detail", "maintain subject details", "marks entry" etc.

This approach lead to a style of design where we decide how to combine these modules to provide larger ones; to combine those to provide even larger ones, and so on, till we arrive at one big module which is the whole of the desired program. The set of these modules form a hierarchy as shown in Fig. 5.13. This is a cross-linked tree structure in which each module is subordinate to those in which it is used.

Since the design progressed from bottom layer upwards, the method is called bottom-up design. The main argument for this design is that if we start coding a module soon after its design, the chances of recoding is high; but the coded module can be tested and design can be validated sooner than a module whose sub modules have not yet been designed.

This method has one terrible weakness; we need to use a lot of intuition to decide exactly what functionality a module should provide.

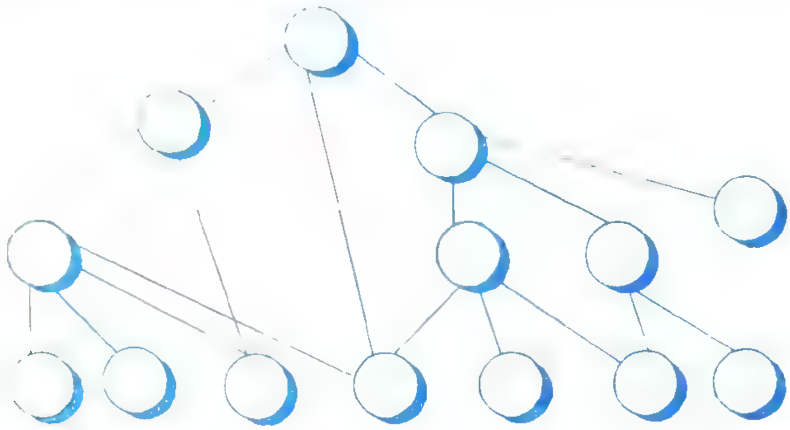


Fig. 5.13: Bottom-up tree structure

If we get it wrong, then at a higher level, we will find that it is not as per requirements; then we have to redesign at a lower level. If a system is to be built from an existing system, this approach is more suitable, as it starts from some existing modules.

5.3.2 Top-Down Design

The essential idea of top-down design is that the specification is viewed as describing a black box for the program? The designer should decide how the internals of the black box is constructed from smaller black boxes; and that those inner black boxes be specified. This process is then repeated for those inner boxes, and so on till the black boxes can be coded directly.

A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. Most design methodologies are based on this approach and this is suitable, if the specifications are clear and development is from the scratch. If coding of a part starts soon after it's design, nothing can be tested until all its subordinate modules are coded.

5.3.3 Hybrid Design

Pure top-down or pure bottom-up approaches are often not practical. For a bottom-up approach to be successful, we must have a good notion of the top to which the design should be heading. Without a good idea about the operations needed at the higher layers, it is difficult to determine what operations the current layer should support [JALO98].

For top-down approach to be effective, some bottom-up (mostly in the lowest design levels) approach is essential for the following reasons:

- To permit common sub modules
- Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more numbers of modules at low levels than at high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

Hybrid approach has really become popular after the acceptance of reusability of modules. Standard libraries, microsoft foundation classes (MFCs), object oriented concepts are the steps in this direction. We may soon have internationally acceptable standards for reusability.

5.4 FUNCTION ORIENTED DESIGN

The design activity begins when the SRS document for the software to be developed is available. The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, their specifications of these modules, and how the modules should be interconnected.

Function oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional viewpoint.

One of the best-known advocates of this method is Niklaus Wirth, the creator of PASCAL and a number of other languages. His special variety is called stepwise refinement, and it is a top down design method. We start with a high level description of what the program does. Then, in each step, we take one part of our high level description and refine it, i.e. specify in somewhat greater detail what that particular part does.

This method works fine for small programs. For large programs its value is more questionable. The main problem is that it is not easy to know what a large program does. For instance, what does UNIX do? Or an airline reservation system? Or a scheme interpreter? The answer is that it depends on what the user types at the terminal. Still, one can usually come up with some kind of high-level function. The risk is that this function is a highly artificial description of reality.

Consider the example of scheme interpreter. Top-level function may look like:

```
While (not finished)
{
        Read an expression from the terminal;
        Evaluate the expression;
        Print the value;
}
```

We thus get a fairly natural division of our interpreter into a "read" module, an "evaluate" module and a "print" module. Now we consider the "print" module and is given below:

```
        Print (expression exp)
{
        Switch (exp → type)
        Case integer: /*print an integer*/
        Case real:    /*print a real*/
        Case list:    /*print a list*/
        :::
}
```

The other modules are structured in a similar way. We notice that the different kinds of objects that are to be manipulated by the scheme interpreter (integer, real, etc.) need to be known by every module. Thus, if we need to add a type, every module needs to be altered. Needless to say, we would like to avoid that.

We continue the refinement of each module until we reach the statement level of our programming language. At that point, we can describe the structure of our program as a tree of refinements as in design top-down structure as shown in Fig. 5.14.

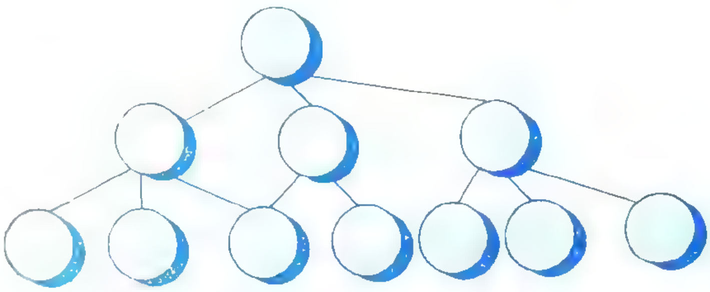


Fig. 5.14: Top-down structure

Unfortunately, if a program is created top-down, the modules become very specialized. As one can easily see in top down design structure, each module is used by at most one other module, its parent. For a module to be reusable, however, we must require that several other modules as in design-reusable structure as shown Fig. 5.15.

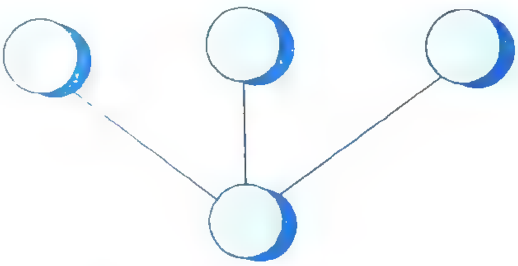


Fig. 5.15: Design reusable structure

It is, of course, not necessary to create a program top-down, even though its structure is function-oriented. However, if we want to delay the decision of what the system is supposed to do as long as possible, a better choice is to structure the program around the data rather than around the actions taken by the program.

5.4.1 Design Notations

During the design phase there are two things of interest: the design of the system, and the process of designing itself. It is for the latter that principles and methods are needed.

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. For a function oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts
- Pseudocode

The first two techniques have been discussed in chapter 3 and other two are discussed in this section.

Structure chart

The structure chart is one of the most commonly used method for system design. It partition a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design. Inputs are given to black box and appropriate outputs are generated by the black box. This concept reduces the complexity because details are hidden from those whose have no need or desire to know. Thus, systems are easy to construct and easy to maintain. Here, black boxes are arranged in hierarchical format as shown in Fig. 5.16.

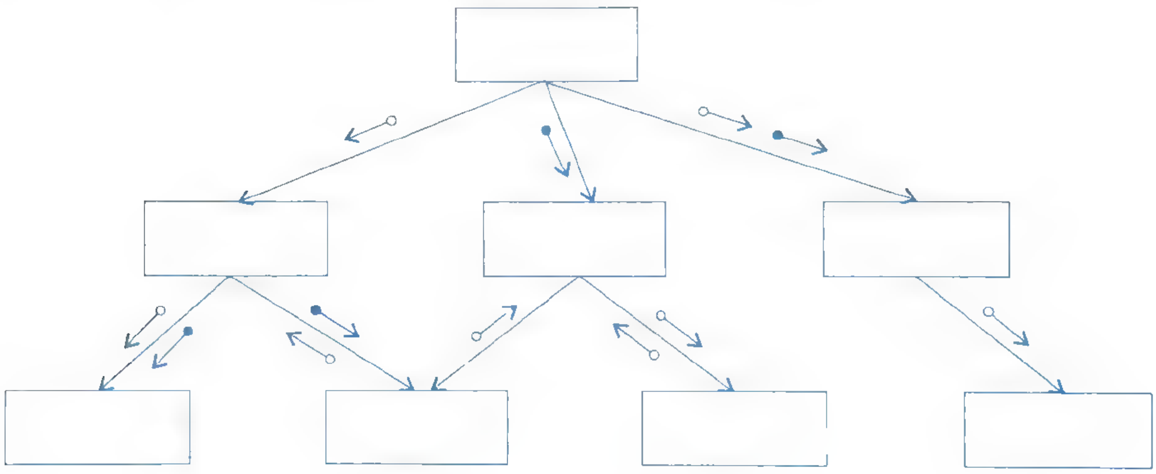


Fig. 5.16: Hierarchical format of a structure chart

In a structure chart, each program module is represented by a rectangular box. Modules at the top level call the modules at the lower level. The connection between modules are represented by lines between the rectangular boxes. The components are generally read from top to bottom, left to right. Modules are numbered in hieratical numbering scheme.

When a module calls another, it views the called module as a black box, passing parameters needed for the called module's functionality and receiving answers. Control data passed between modules on a structure chart are represented by labelled directed arrow with filled in circle and data is depicted with an open circle. When a module is used by many other modules, it is put into the library of modules. The diamond symbol is used to represent the fact that one module out of several modules connected with the diamond symbol is used depending on the outcome of the condition attached to the diamond symbol. A loop around the control flow arrows denotes that the respective modules are used repeatedly and is called repetition symbol. Fig. 5.17 shows the notations used in structure chart:

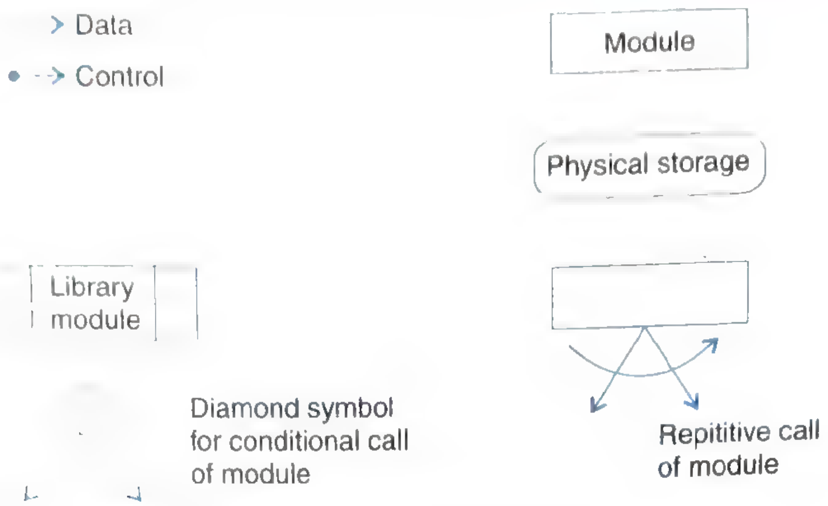


Fig. 5.17: Structure chart notations

A structure chart for "update file" is given in Fig. 5.18.

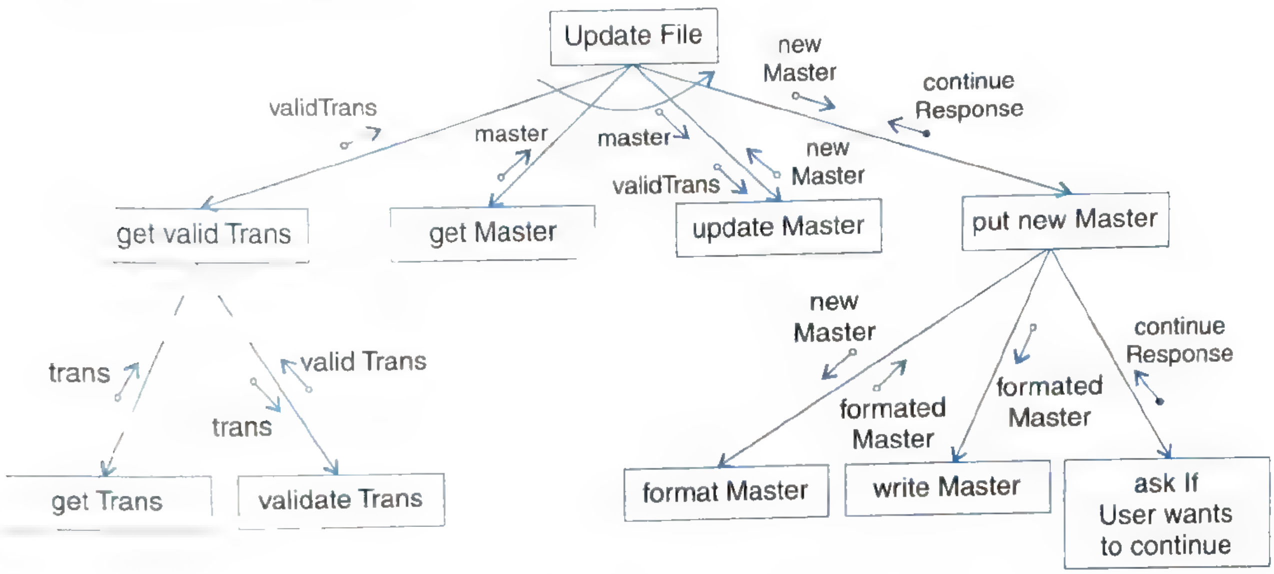


Fig. 5.18: Update file

The 'Update file' calls 'get Valid Trans' to get input parameter Valid Trans. Module 'get valid tans' calls 'get trans' module to read the trans from the input device and pass it back to 'get Valid Trans'. 'Get Valid Trans' then calls validate Trans 'module' to validate the transaction which is subsequently passed to the 'update file' module. 'Update file' module then calls the get master and passes the master and 'valid trans' information to 'update master'. 'Update' master module passes the new master data to 'update file'.

'Update file' then involves 'put new master' by passing 'new master' data to it. Put new Master involves format master 'write Master' and ask if user wants to continue Module 'Continue Resource control data' is passed to 'update file' to decide user wants to continue by repeating the above procedure.

This type of structure chart is often called as transform-centred structures. Transform-centered structure clout receive an input which is transformed by a sequence of operations, with each operation being carried out by one module. Fig. 5.19 above is an example of transform-

centred structures. Another common type of structure is transaction centred structure. A transaction centred structure describes a system that processes a number of different types of transactions. It is illustrated in Fig. 5.19.

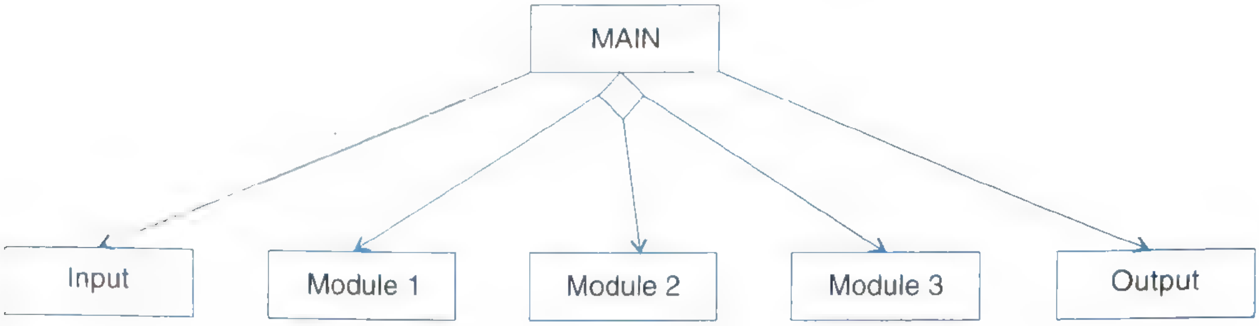


Fig. 5.19: Transaction-centered structure

In the above figure the MAIN module controls the system operation its function is to:

- invoke the INPUT module to read a transaction;
- determine the kind of transaction and select one of a number of transaction modules to process that transaction, and
- output the results of the processing by calling OUTPUT module.

Pseudocode

Psudocode notation can be used in both the preliminary and detailed design phases. Like flowcharts, psudocode can be used at any desired level of abstraction. Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as If-Then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Using the top-down design strategy, each English phrase is expanded into more detailed pseudocode until the design specification reaches the level of detail of the implementation language.

Pseudocode can replace flowcharts and reduce the amount of external documentation required to describe a system [FAIR2K].

5.4.2 Functional Procedure Layers

- Functions are built in layers, Additional notation is used to specify details.
- Level 0
  - Function or procedure name
  - Relationship to other system components (e.g., part of which system, called by which routines, etc.)
  - Brief description of the function purpose.
  - Author, date.
- Level 1
  - Function parameters (problem variables, types, purpose, etc.)
  - Global variables (problem variable, type, purpose, sharing information)
  - Routines called by the function.

- Side effects.
- Input/Output Assertions.

- Level 2
  - Local data structures (variable etc.)
  - Timing constraints
  - Exception handling (conditions, responses, events)
  - Any other limitations.
- Level 3
  - Body (structured chart, English pseudo code, decision tables, flow charts, etc.)

5.5 IEEE RECOMMENDED PRACTICE FOR SOFTWARE DESIGN DESCRIPTIONS (IEEE STD 1016–1998)

5.5.1 Scope

This is a recommended practice for describing software designs. This is designed by IEEE (Institution of Electricals and Electronics Engineers) and is known as IEEE Standard (IEEE std. 1016–1998) for software design description (SDD). An SDD is a representation of a software system that is used as a medium for communicating software design information.

5.5.2 References

This standard shall be used in conjunction with the following publications.

(*i*) IEEE std 830–1998, IEEE recommended practice for software requirements specifications

(*ii*) IEEE std 610.12–1990, IEEE glossary of software engineering terminology.

5.5.3 Definitions

Few important definitions are given below:

(*i*) **Design entity:** An element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

(*ii*) **Design View:** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.

(*iii*) **Entity attribute:** A named property or characteristics of a design entity. It provides a statement of fact about the entity.

(*iv*) **Software design description (SDD):** A representation of a software system created to facilitate analysis, planning, implementation and decision making. A blue print or model of the software system. The SDD is used as the primary medium for communicating software design information.

5.5.4 Purpose of an SDD

The SDD shows how the software system will be structured to satisfy the requirements identified in the SRS. It is basically the translation of requirements into a description of the software

structure, software components, interfaces, and data necessary for the implementation phase. Hence, SDD becomes the blue print for the implementation activity.

5.5.5 Design Description Information Content

(*i*) **Introduction:** The SDD is a representation or model of the system to be created. The model should provide the precise design information needed for planning and implementation of the software system. It should represent partitioning of the system into design entities and describe the important properties and relationship among those entities.

(*ii*) **Design entities:** A design entity is an element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

Design entities result from a decomposition of the software system requirements. The objective is to divide the system into separate components that can be considered, implemented, changed, other components.

The entities may have different nature, but may have common characteristics. Each design entity will have a name, purpose, and function. There are common relationship among entities such as interfaces or shared data. The common characteristics of entities are described by design entity attributes.

(*iii*) **Design entity attributes:** A design entity attribute is a named characteristic or property of a design entity. It provides a statement of fact about the entity.

Design attributes can be thought of as questions about design entities. The answer to those questions are the values of the attributes. All the questions can be answered, but the content of the answer will depend upon the nature of the entity. The collection of answers provides a complete description of an entity.

All attributes shall be specified for each entity. Attribute descriptions should include references and design considerations such as tradeoffs and assumptions when appropriate. In some cases, attribute descriptions may have the value none. When additional attributes are identified for a specific software project, they should be included in the design description. The attributes and associated information items are defined in the following subsections (from '*a*' to '*j*').

(*a*) ***Identification:*** The name of the entity: Two entities shall not have the same name. The names for the entities may be selected to characterise their nature. This will simplify referencing and tracking in addition to providing identification.

(*b*) ***Type:*** A description of the kind of entity. The type attribute shall describe the nature of entity. It may simply name the kind of entity, such as subprogram, module, procedure, process, or data store. Alternatively, design entities may be grouped into major classes to assist in locating an entity dealing with a particular type of information. For a given design description, the chosen entity types shall be applied consistently.

(*c*) ***Purpose:*** A description of why the entity exists. The purpose attribute shall provide the rationale for the creation of the entity. Therefore, it shall designate the specific functional and performance requirements for which the entity is created.

(*d*) ***Function:*** A statement of what the entity does. The function attribute shall state the transformation applied by the entity to inputs to produce the desired output. In the case of

a data entity, this attribute shall state the type of information stored or transmitted by the entity.

(e) ***Subordinates:*** The identification of all entities composing this entity. The subordinates attribute shall identify the "composed of relationship" for an entity. This information is used to trace requirements to design entities and to identify parent/child structural relationships through a software system decomposition.

(f) ***Dependencies:*** A descriptions of the relationships of this entity with other entities. The dependencies attribute shall identify the uses or requires the presence of relationship for an entity. These relationships are often graphically depicted by structure charts, data flow diagrams etc.

(g) ***Interface:*** A description of how other entities interact with this entity. The interface attribute shall describe the methods of interaction and the rules governing those interactions. The methods of interaction include the mechanisms for invoking or interrupting the entity, for communicating through parameters, common data areas or messages, and for direct access to internal data. The rules governing the interaction include the communications protocol, data format, acceptable values, and the meaning of each value.

This attribute shall provide a description of the input ranges, the meaning of inputs and outputs, the type and format of each input or output error codes. For information systems, it should include inputs, screen formats, and a complete description of the interactive language.

(h) ***Resources:*** A description of the elements used by the entity that are external to the design. The resources attribute shall identify and describe all of the resources external to the design that are needed by this entity to perform its function. The interaction rules and methods for using the resource shall be specified by this attribute.

This attribute provides information about items such as physical devices (like printers), software services (like math libraries, OS services), and processing resources (like CPU cycles, memory allocation, buffers).

(i) ***Processing:*** A description of the rules used by the entity to achieve its function. The processing attribute shall describe the algorithm used by the entity to perform a specific task and shall include contingencies. This description is a refinement of the function attribute.

(j) ***Data:*** A description of data elements internal to the entity. The data attribute shall describe the method of representation, initial values, use, semantics, format, and acceptable values of internal data.

The description of data may be in the form of a data dictionary that describes the content, structure, and use of all data elements. Data information shall describe everything pertaining to the use of data or internal data structures by this entity. It shall include data specifications such as formats, number of elements, and initial values. It shall also include the structures to be used for representing data such as file structures, arrays, stacks, queues, and memory partitions.

5.5.6 Design Description Organisation

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organisation of SDD is given in Table 5.1. This is one of the possible ways to organise and format the SDD.

Table 5.1: Organisation of SDD

1. Introduction
    1.1 Purpose
    1.2 Scope
    1.3 Definitions and acronyms
2. References
3. Decomposition description
    3.1 Module decomposition
        3.1.1 Module 1 description
        3.1.2 Module 2 description
    3.2 Concurrent process decompostion
        3.2.1 Process 1 description
        3.2.2 Process 2 description
    3.3 Data decomposition
        3.3.1 Data entity 1 description
        3.3.2 Data entity 2 description
4. Dependency description
    4.1 Intermodule dependencies
    4.2 Interprocess dependencies
    4.3 Data dependencies
5. Interface description
    5.1 Module Interface
        5.1.1 Module 1 description
        5.1.2 Module 2 description
    5.2 Process interface
        5.2.1 Process 1 description
        5.2.2 Process 2 description
6. Detailed design
    6.1 Module detailed design
        6.1.1 Module 1 detail
        6.1.2 Module 2 detail
    6.2 Data detailed design
        6.2.1 Data entry 1 detail
        6.2.2 Data entry 2 detail

Entity attribute information can be organised in several ways to reveal all of the essential aspects of a design. There may be number of ways to view the design. Hence, each design view represents a separate concern about a software system. Together, these views provide a comprehensive description of the design in a concise and usable form that simplifies information access and assimilation. A recommended organisation of the SDD into separate design views to facilitate information access and assimilation is given in Table 5.2.

Table 5.2: Design views

Design view	*Scope*	*Entity attribute*	*Example representation*
Decomposition description	Partition of the system into design entities.	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

5.6 OBJECT ORIENTED DESIGN

"Object Oriented" has clearly become the buzzword of choice in the industry. Almost everyone talks about it. Almost everyone claims to be doing it, and almost everyone says it is better than traditional function oriented design. Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to be manipulated by the program. Thus, it is orthogonal to function oriented design.

Object Oriented Design begins with an examination of the real world "things" that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes (transient state information) and behaviour (functional process information). Each object maintains its own state, and offers a set of services to other objects. Shared data areas are eliminated and objects communicate by message passing (e.g. parameters). Objects are independent entities that may readily be changed because all state and representation information is held within the object itself. Objects may be distributed and may execute either sequentially or in parallel [BOOC03].

5.6.1 Basic Concepts

Object Oriented Design is not dependent on any specific implementation language. Problems are modelled using objects. Objects have:

- Behaviour (they do things)
- State (which changes when they do things)

For example, a car is an object. It has state: whether its engine is running; and it has a behaviour: starting the car, which changes its state from "engine not running" to "engine running".

The various terms related to object oriented design are Objects, Classes, Abstraction, Inheritance and Polymorphism.

(i) **Objects:** The word "Object" is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behaviour) to either examine or affect this state. Hence, an object is characterised by number of operations and a state which remembers the effect of these operations [JACO 98].

All objects have unique identification and are distinguishable. Two bananas may be of same colour, shape and texture but still are different. Each has an identity. There may be four dogs of same colour, breed and size but all are distinguishable. The term identity means that objects are distinguished by their inherent existence and not by descriptive properties [JOSHO 3].

As discussed above, object is an entity, which has a state (whose representation is hidden from other objects) and a defined set of operations, which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects, which request these services when some computation is required. In principle, objects communicate by passing messages to each other and these messages initiate object operations.

(ii) **Messages:** Conceptually, objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. In some distributed systems, object communications are implemented directly as text messages which objects exchange. The receiving object parses the message, identifies the service and the associated data and carries out the requested service. Messages are often implemented as procedure or function calls (name = procedure name, information = parameter list).

(iii) **Abstraction:** In object oriented design, complexity is managed using abstraction. *Abstraction is the elimination of the irrelevant and the amplification of the essentials.*

We can teach someone to drive any car using an abstraction. We amplify the essentials: we teach about the ignition and steering wheel, and we eliminate the details, such as details of the particular engine in this car or the way fuel is pumped to the engine where a spark ignites it, it explodes pushing down a piston and driving a crankshaft [FIEL01].

Problems usually have levels of abstraction. We can see a car at a high level of abstraction for driving, but mechanics need to work at a lower level of abstraction. They do care about details and need to know about batteries and engines.

However, there are abstractions at the mechanic's level too. The mechanic might test or charge a battery without caring that inside the battery there is a complex chemical reaction going on. A battery designer would care about these details but would not care about, say, the electronics that goes into the car's stereo.

We have seen that we can look at the details such as the battery or stereo design, and they are separated in manageable chunks, and by using abstraction and ignoring the details, we can also look at the whole car as a manageable chunk.

(*iv*) **Class:** In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping in known as a class. Hence, a class is a set of objects that share a common structure and a common behaviour.

We may have a class "car" with objects like Indica, Santro, Maruti, Indigo. These objects are related due to some common characteristics to constitute a class named "car". A class may be object defined as [JACO98]:

"A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structures.

In object oriented system, each object belongs to a class. An objects, that belongs to a certain class is called an instance of that class. We often use object and instance as synonyms. Hence, an instance is an object created from a class. The class describes the structure (behaviour and information) of the instance, while the current state of the instance is defined by the operations performed on the instance.

We may define a class "car" and each object that represents a car becomes an instance of this class. In this class "car", Indica, Santro, Maruti, Indigo are instances of this class as shown in Fig. 5.20.

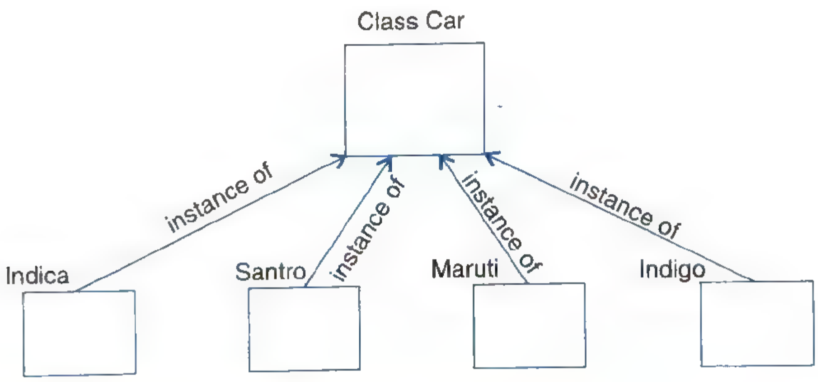


Fig. 5.20: Indica, Santro, Maruti, Indigo are all instances of the class "car"

We may have different types of classes depending upon common characteristics. The class is the static description and the object is an instance in runtime of that class.

Imagine a picture made up of squares. Each square is an object. It has a state: its colour and position, and behaviour. We can, amongst other things, change its colour and draw it. Each square is different but has much in common with other squares. So, we abstract out the commonalities: they share the same behaviour and have same sort of attributes. We have ignored same sort of attributes. We have ignored the particular values of the attributes.

Classes are useful because they act as a blue print for objects. If we want a new square we may use the square class and simply fill in the particular details (*i.e.*, colour and position). Fig. 5.21 shows how can we represent the square class.

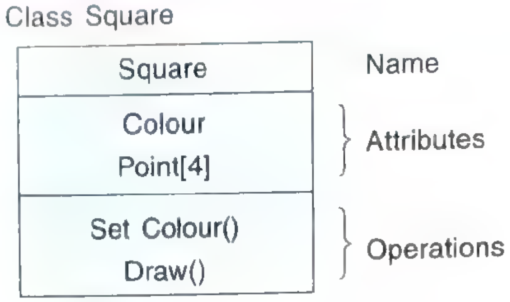


Fig. 5.21: The square class

(*v*) **Attributes:** An attribute is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attribute has a value for each object instance. For example, colour may be different in different objects and "array of points" size may also be different in different squares. The attributes are shown as second part of the class as shown in Fig. 5.21.

(*vi*) **Operations:** An operation is a function or transformation that may be applied to or by objects in a class. In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. Each operation has a target object as an implicit argument. The behaviour of the operation depends on the class of its target. An object "knows" its class, and hence the right implementation of the operation [JOSHO3]. Operations are shown in the third part of the class as indicated in Fig. 5.21.

(*vii*) **Inheritance:** Imagine that, as well as squares, we have triangle class. Fig. 5.22 shows the class for a triangle.

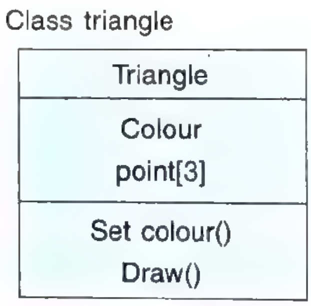


Fig. 5.22: The triangle class

Now, comparing Fig. 5.21 and Fig. 5.22, we can see that there is some difference between triangle and squares classes. Triangles have three vertices ; squares have four. Also, the way that these shapes are drawn is different. However, there are some similarities. For example, we can set the colour of both and both can be drawn (even if the way they are drawn is different). It would be nice if we could abstract; eliminate the details of each shape and amplify the fact that both can have their colours set and can be drawn. For example, at a high level of abstraction, we might want to think of a picture as made up of shapes and to draw the picture, we draw each shape in turn. We want to eliminate the irrelevant details: we do not care that one shape is a square and the other is a triangle as long as both can draw themselves.

To do this, we consider the important parts out of these classes in to a new class called Shape. Fig. 5.23 shows the results.

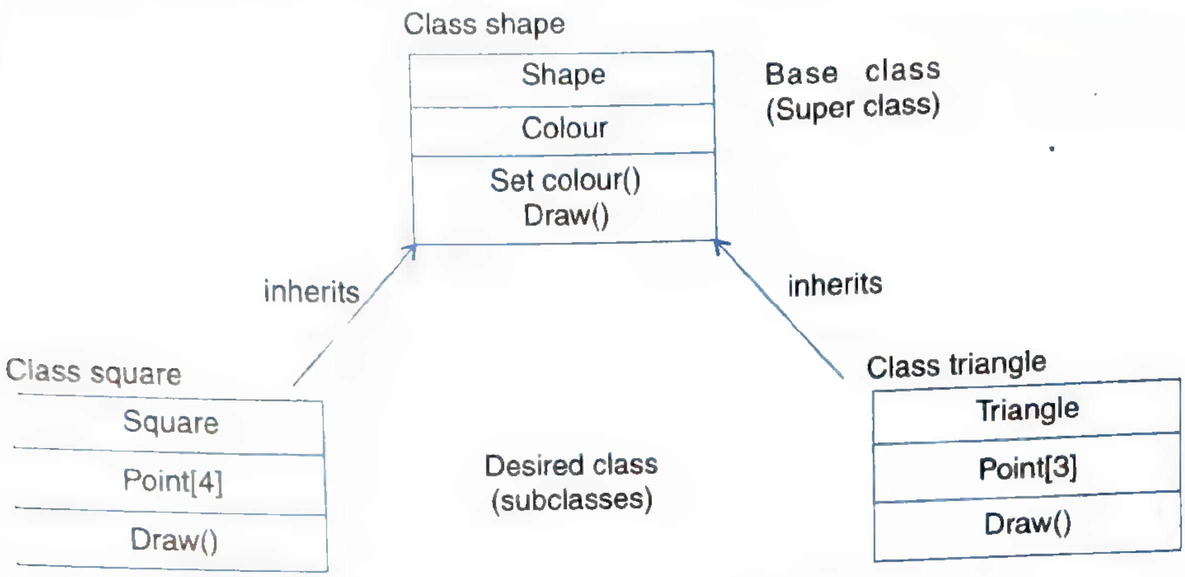


Fig. 5.23: Abstracting common features in a new class

This sort of abstraction is called inheritance. The low level classes (known as subclasses or derived classes) inherit state and behaviour from this high level class (known as a super class or base class).

Hence, a triangle object will have a colour, a setcolour() operation, three point and a draw() operation. We can inherit three sorts of things:

(*i*) State: colour

(*ii*) Operations: setcolour() should be able to set the colour for any shape.

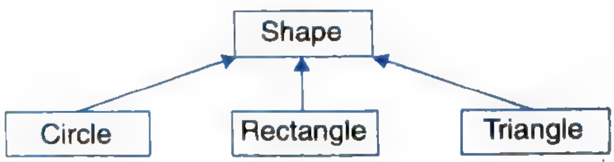
(*iii*) The interface of an operation.

Shape contains the interface of the draw() operation because draw() interface for triangle and square is identical but code for the operation remains in the subclasses because it is different.

(*viii*) **Polymorphism:** When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

So, we can abstract by pulling out important state, behaviour and interface into an new class. We can also abstract by combining object's inside a new object. In the car example, a car combines a battery, engine and other objects into a new object and provides a simple interface for driving that hides these details. There are different sorts of abstraction and finding the best ways to apply abstraction to a problem is what design is all about.

Another example may be for a base class *shape,* polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the *area* method to it will return the correct results. Plolymorphism is considered to be a requirement of any true object-oriented programming language (OOPL).



(ix) **Encapsulation (Information Hiding):** Encapsulation is also commonly referred to as "Information Hiding. It consists of the separation of the external aspects of an object from the internal implementation details of the object. The external aspects of an object are accessible by other objects through methods of object, while the internal implementation of those methods are hidden from the external object sending the message. Thus, it is possible to change the implementation without updating the clients as long as the interface is unchanged. Clients will not be affected by changes in implementation, thus reducing the "ripple effect" in which a correction to one operation forces the corresponding correction in a client operation which in turn causes a change in a client of the client. This makes maintenance is easier and less expensive.

Encapsulation deals with permitting or restricting a client class' ability to modify the attributes or invoke the methods of the class or object of concern. If a class allows another to modify its attributes or invoke its methods, the attributes and methods are said to be part of the class' public interface. If a class doesn't allow another to modify its attributes or invoke its methods, those are part of the class' private implementation. Thus, Encapsulation protects (a) an object's internal state from being corrupted by its clients and (b) Client code from changes in the object's implementation.

A "Queue" provides a good example of this characteristic. A queue is an abstract concept that represents an ordered list of things. The implementation of a queue may by an array or it may be by a linked-list. If the implementation were known, a developer writing a client of the queue class may use this knowledge and directly access the internal storage mechanism. If the implementation changed, the client would then have to be modified also. This type of tight coupling between classes would cause a very brittle system and would increase maintenance costs as parts of the system were modified. Therefore, the levels of encapsulation that a language supports and how those mechanisms are used directly impacts the level of coupling between classes and it can significantly affect the cost of maintenance in an application.

(x) **Hierarchy:** Hierarchy involves organizing something according to some particular order or rank (*e.g.*, complexity, responsibility, etc.). It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way. This hierarchy is implemented in software via a mechanism called "Inheritance". Just as a child inherits genes from its parent, a class can inherit attributes and behaviours from its parent. The parent class is commonly referred to as the supper-class and the child class as the sub-class. *Classes at the same level of the hierarchy should be at the same level of abstraction.*

Using a hierarchy to describe differences or variations of a particular concept provides for more descriptive and cohesive abstractions, as well as a better allocation of responsibility. In any one system, there may be multiple abstraction hierarchies (*e.g.*, in a financial application, you may have different types of customers and accounts). Generalization can be used to realize a hierarchy within an object-oriented system, starting at the most general of the abstractions, and then defining more specialized abstractions through sub-classing. Generalization can take place in several stages, which lets you model complex, multilevel inheritance hierarchies. General properties are placed in the upper part of the inheritance hierarchy, and

special properties lower down. In other words, you can use generalization to model specialization of a more general concept.

This relationship can be easily understood by the Fig. 5.24 below.

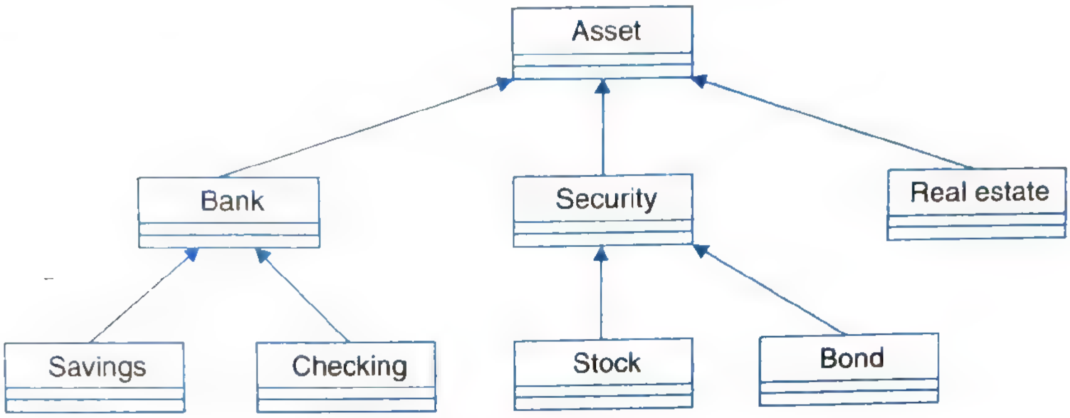


Fig. 5.24: Hierarcy

In summary, with object-oriented design, we use abstraction to break a problem into manageable chunks. We can comprehend the problem as a whole or study parts of the problem at lower level of abstraction.

If we work at an appropriate level of abstraction then any problems we find can be solved relatively easily. As an example of not working at the right level of abstraction, imagine that we tried to build a house by going out with some bricks and just building. The chances are we would finish, try to put the bath in and discover that the bathroom is too small. So, we have to knock down and rebuild a wall. Meaning, thereby, lot of work, assuming it could be done. If we had designed the house including, where the fittings were going. We would have noticed the problem and fixing it would have taken about 30 seconds with our eraser and pencil.

5.6.2 Steps to Analyze and Design Object Oriented System

There are various steps in the analysis and design of an object oriented system and are given in Fig. 5.25.

(*i*) **Create use case model:** First step is to identify the actors interacting with the system. We should then write the use case and draw the use case diagram.

(*ii*) **Draw activity diagram (If required):** Activity diagram illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. It is essentially like a flow chart. Fig. 5.26 shows the activity diagram processing an order to deliver some goods.

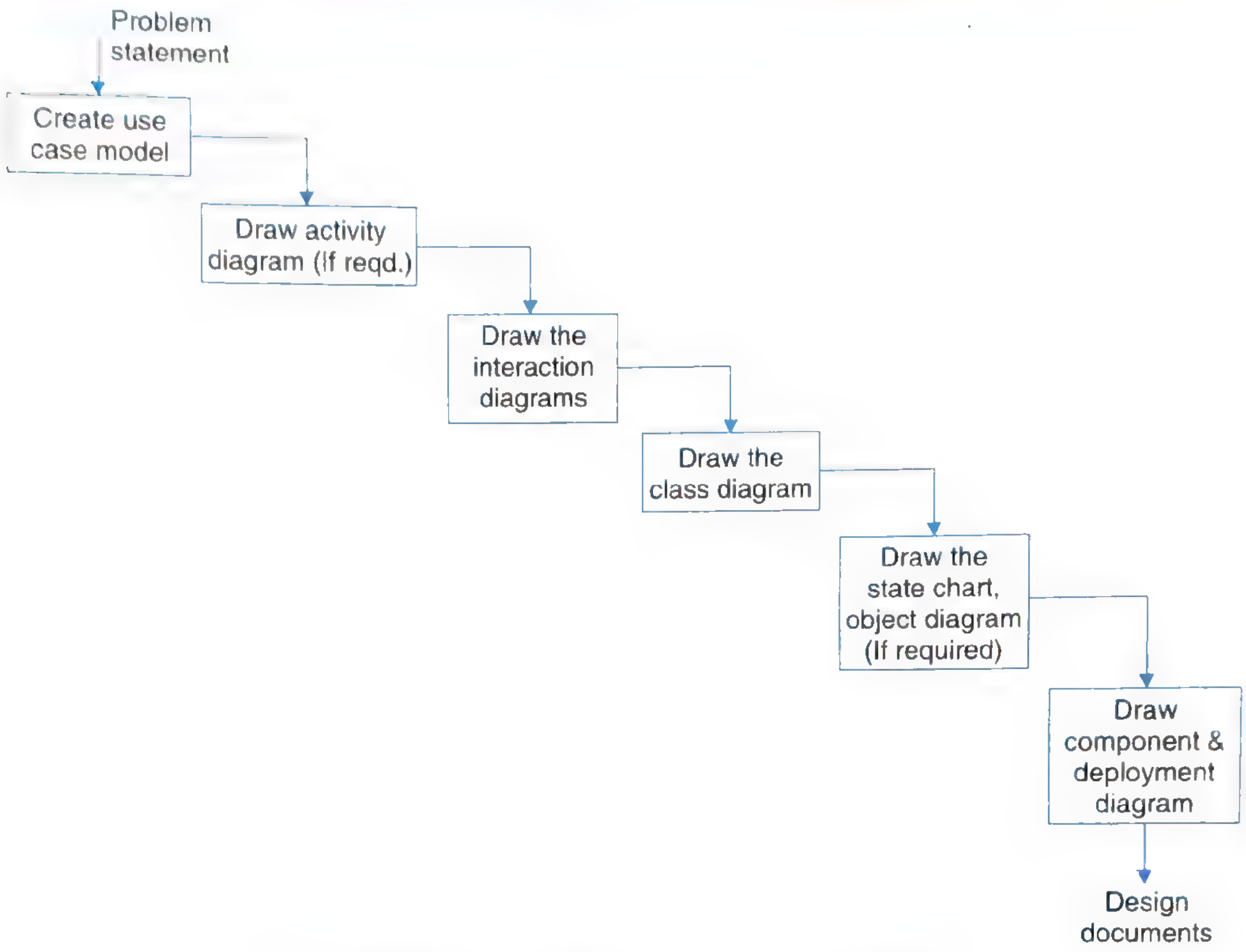


Fig. 5.25: Steps for analysis & design of object oriented system

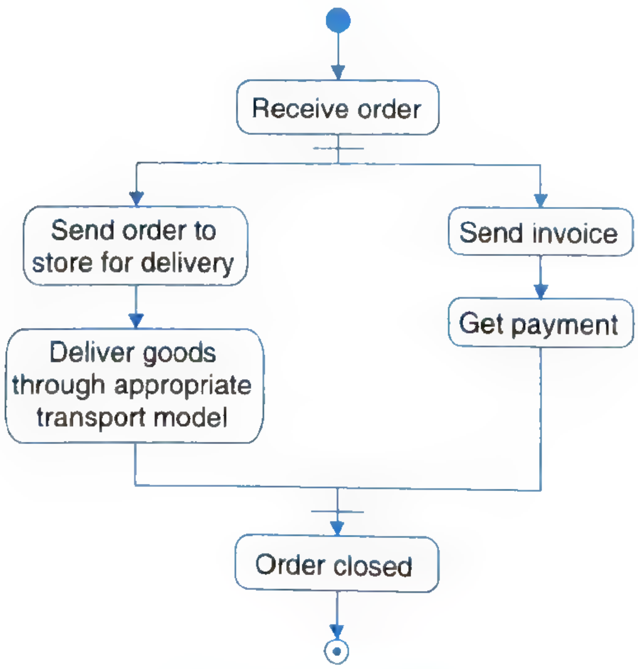


Fig. 5.26: Activity diagram

(*iii*) **Draw the interaction diagram:** An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time ordering of messages; a collaboration diagram is an interaction diagram that emphasizes the structural organisation of the objects that send and receive messages.

Sequence diagrams and collaboration diagrams are isomorphic, meaning that we can take one and transform it into the other.

Steps to draws interaction diagrams are as under :

(*a*) Firstly, we should identify that the objects with respects to every use case.

(*b*) We draw the sequence diagrams for every use case.

(*c*) We draw the collaboration diagrams for every use case.

Many object analysis techniques define only one type of object, which can be used any where in the system. Jacobson *et.al* [JACO98] have chosen to use three object types. The reason for this is to have a structure that is more flexible and adaptable to changes. The object types used is this analysis model are entity objects, interface objects and control objects as given in Fig. 5.27.

Fig. 5.27: Object types

The entity object models information in the system that should be held for a longer time, and should typically survive a use case. All behaviour naturally coupled in this information should be placed in the entity object. For a login use case, login-detail may be the entity object. The interface object models behaviour and information that is dependent on the interface to the system. Thus everything concerning any interface to the system is placed in an interface object. For example login screen of the login use case.

(*iv*) **Draw the class diagram:** The class diagram shows the relationship amongst classes. There are four types of relationships in class diagrams.

(*a*) ***Association*** are semantic connection between classes. When an association connects two classes, each class can send messages to the other in a sequence or a collaboration diagram. Associations can be bi-directional or unidirectional.

(*b*) ***Dependencies*** connect two classes. Dependencies are always unidirectional and show that one class, depends on the definitions in another class.

- - - - ->

(*c*) ***Aggregations*** are stronger form of association. An aggregation is a relationship between a whole and its parts.

(*d*) ***Generalizations*** are used to show an inheritance relationship between two classes.

We may design the class diagram of a complete system. Obviously, this may become very huge, for complex systems. In such a case, one class diagram is made for each use case. This restricts the size of the class diagram and is useful when delegating/assigning the task to a software developer ; since he/she would only need to see the related class diagram of the assigned use case; rather than looking at the complete class diagram and then filtering out the concepts related to his/her use-case. We may also draw object diagram, which shows a set of objects and their relationships. They represent static snapshots of instances of the things found in class diagrams, from the perspective of real cases.

(*v*) ***Design of state chart diagrams:*** A state chart diagram is used to show the state space of a given class, the event that cause a transition from one state to another, and the actions that result from a state change. It shows the changing behaviour of an object in response to different events. State transition diagrams are made only for the objects that are either very complicated or have a very dynamic behaviour with respect to various events. Normally, we would not need to make state diagrams. A state transition diagram for a "book" in the library system is given in Fig. 5.28.

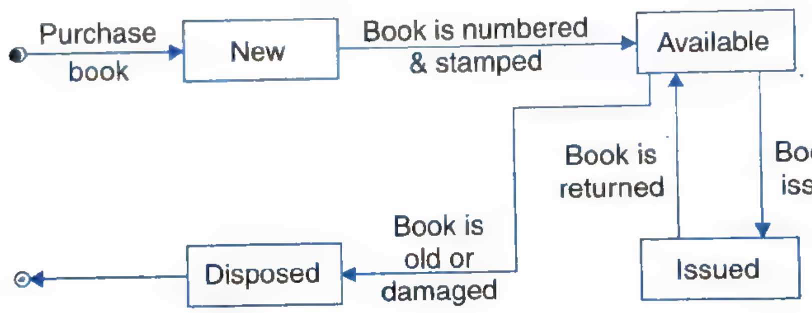


Fig. 5.28: Transition chart for 'book' in a library system

(*vi*) ***Draw component and development diagram:*** Component diagrams address the static implementation view of a system they are related to class diagrams in that a component typically maps to one or more classes, interfaces or collaboration. Deployment diagram captures relationship between physical components and the hardware.

After the completion of above mentioned steps, we will have many documents and complete understanding of flow of data, control and relationships of classes. These things are the foundations for implementing an object oriented system.

5.6.3 Case Study of Library Management System

The problem statement for library management system is given below:

Problem statement

A software has to be developed for automating the manual library of a University. The system should be stand alone in nature. It should be designed to provide functionality's as explained below:

Issue of books

- A student of any course should be able to get books issued.
- Books from general section are issued to all but book bank books are issued only for their respective courses.
- A limitation is imposed on the number of books a student can issue.
- A maximum of 4 books from book bank and 3 books from general section is issued for 15 days only.
- The software takes the current system date as the date of issue and calculates date of return.
- A bar code detector is used to save the student as well as book information.
- The due date for return of the book is stamped on the book.

Return of books

- Any person can return the issued books.
- The student information is displayed using the bar code detector.
- The system displays the student details on whose name the books were issued as well as the date of issue and return of the book.
- The system operator verifies the duration for the issue.
- The information is saved and the corresponding updating take place in the database.

Query processing

The system should be able to provide information like:

- Availability of a particular book.
- Availability of book of any particular author.
- Number of copies available of the desired book.

The system should also be able to generate reports regarding the details of the books available in the library at any given time. The corresponding printouts for each entry (issue/return) made in the system should be generated. Security provisions like the 'login authenticity should be provided. Each user should have a user id and a password. Record of the users of the system should be kept in the log file. Provision should be made for full backup of the system.

Use cases

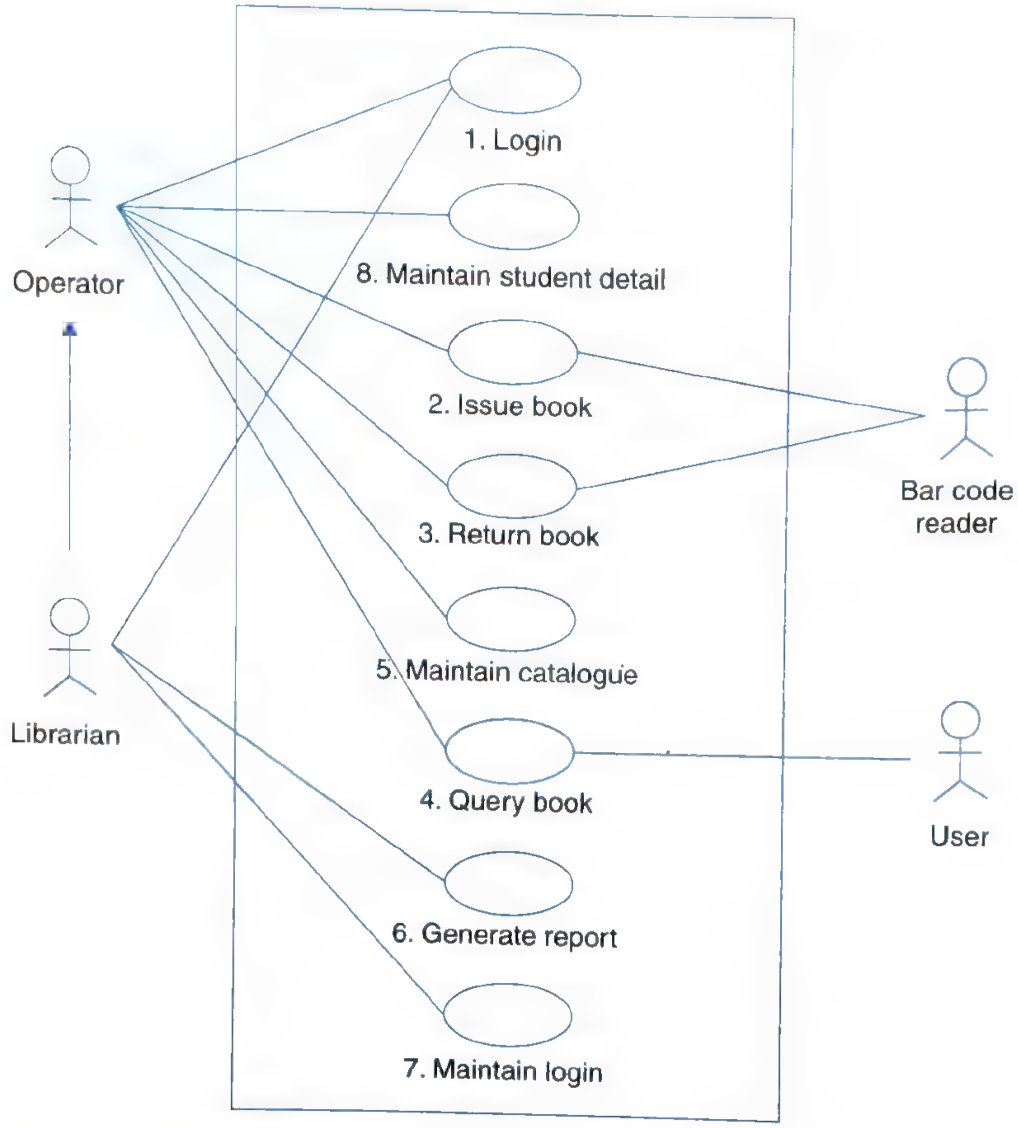
From the problem description, we can see that the system has four actors.

1. Librarian
2. Operator
3. Bar Code Reader
4. User

To define system's functionalities, we can view the system as a collection of following use cases:

1. Login
2. Issue Book
3. Return Book
4. Query Book

5. Maintain Catalog
6. Generate Report
7. Maintain Login
8. Maintain student Details



Use case diagram for library management system

USE CASE DESCRIPTION

1. LOGIN

1.1. Introduction

This use case documents the pro into the Library Management System based on user privileges.

- Operator (Issue Book, Return Book, Query a Book, Maintain Catalogue, Maintain Student Detail)
- Librarian (Generate Reports, Maintain Login)

1.2 Actors

Operator, Librarian

1.3 Pre-Condition

None

1.4 Post-Condition

If use case is successful, the user is logged into the system, otherwise the system state is unchanged.

1.5 Flow of Events

1.5.1 Basic Flow

This use case starts when actor wishes to log in to the Library Management System.

- The system requests that the actor enters his/her user_id and password.
- The actor enters user_id and password.
- The system validates the user_id and password and checks for his/her privileges.
- If the user is "operator", he/she will be logged into the system and presented with operator's menu.
- Otherwise, if the user is "librarian", he/she will be logged into the system and presented with librarian's menu.
- The use case ends.

1.5.2 Alternate Flow

1.5.2.1 Invalid Name / Password

If the system receives an invalid user_id or password, and error message is displayed and the use case ends.

1.6 Special Requirements

None

1.7 Related Use Cases

None

2. ISSUE BOOK

2.1 Introduction

This use case documents the procedure of issuing a book for following accounts:

- General (for 15 days)
- Book Bank (for the semester)

2.2 Actors

Operator, Barcode reader

2.3 Pre-Condition

Operator must be logged in to the system

2.4 Post-Condition

If use case is successful, the book is issued to the student in his/her general or book bank account, otherwise the system state is unchanged.

2.5 Flow of Events

2.5.1 Basic Flow

The use case starts when a student wants to get a book issued.

- The system reads and validates the student's information using the Bar Code Reader.
- The system reads the book's information using the Bar Code Reader.
- The return date of the book is calculated as per the account in which the student wishes to get the book issued – 15 days for General account and whole semester for Book Bank account.
- The book and student's information is saved into the database.
- The issue details are sent to the printer to generate the receipt.
- The use case ends.

2.5.2 *Alternate Flow*

2.5.2.1 Unauthorized Student

If the system doesn't validates the student, then an error message is flagged and the use case ends.

2.5.2.2 General Account is Full

If the student has requested a book in General Account and the later is full, *i.e.* he has already 3 books issued on his name, then the request for issue is denied and the use case ends.

2.5.2.3 Book Bank Account is Full

If the student has requested a book in Book Bank Account and the later is full, *i.e.* he has already 4 books issued on his name, then and error message is shown and the use case ends.

2.5.2.4 Course Mismatch between Student and the Book Bank Book.

If the student of a particular course has requested a book from some other course on Book Bank Account, the an option for getting the book issued in general account is given and the use case ends.

2.6 Special Requirements

None

2.7 Related Use Cases

Generate Barcode

3. RETURN BOOK

3.1 Introduction

This use case documents the procedure of returning a book and calculating the fine amount if the students has returned the book after the specified return date.

3.2 Actors

Operator, Barcode Reader

3.3 Pre-Condition

Operator must be logged in to the system.

3.4 Post-Condition

If use case is successful, the book is returned back to the library and if needed, the fine is calculated, otherwise the system state is unchanged.

3.5 Flow of Events

3.5.1 Basic Flow

This use case starts when a student wants to return a book.

- The system reads the book's information using the Bar Code Reader.
- The book is returned to the library.
- The database entries corresponding both to the student account and the book are updated.
- The return details are sent to the printer to generate the receipt.
- The use case ends.

3.5.2 Alternate Flow

3.5.2.1 Late return of book

If the book is returned after the due date, fine is calculated and database is updated accordingly. The use case end here.

3.6 Special Requirements

None

3.7 Related Use Cases

Generate Barcode

4. QUERY A BOOK

4.1 Introduction

This use case documents the procedure for searching a book based on the specified criteria, which are:

- Search by Author Name
- Search by Title Name

4.2 Actors

Operator, user

4.3 Pre-Condition

Operator user must be logged in to the system

4.4 Post-Condition

If use case is successful, the book details are displayed.

4.5 Flow of Events

4.5.1 Basic Flow

This use case starts when a student wants to search for a particular book.

- The system displays the various search criteria to the user.
- The user selects the search criteria.
- The result is displayed to the user.
- The use case ends.

4.6 Special Requirements

None

4.7 Related Use Cases

None

5. MAINTAIN CATALOG

5.1 Introduction

This use case documents the procedure for updating the catalog of the library.

5.2 Actors

Operator

5.3 Pre-Condition

Operator must be logged into the system.

5.4 Post-Condition

If use case is successful, the book should be updated, otherwise the system state in unchanged.

5.5 Flow of Events

5.5.1 Basic Flow

This use case starts when the operator wishes to add, delete or modify some details in the library.

- The corresponding changes are saved in the database.
- The use case ends.

5.5.2 Alternate Flow

None

5.6 Special Requirements

None

5.7 Related Use Cases

None

6. GENERAL REPORTS

6.1 Introduction

This use case documents the procedure for generating the reports as desired by the Librarian.

6.2 Actors

Librarian

6.3 Pre-Condition

Librarian must be logged into the system.

6.4 Post-Condition

If use case is successful, the various reports, regarding the details of the books available in the library at any given time, are generated.

6.5 Flow of Events

6.5.1 Basic Flow

This use case starts when a librarian wants to generate reports of the books available in the library.

- The system displays the various report generating criteria to the user, which can be the books issued to the students at a particular time, books available in the library etc.

- The librarian selects the criteria and enters the various parameters based on the criteria selected.
- The system generates the report and sends that to printer.
- The use case ends.

6.5.2 Alternate Flow

6.5.2.1 Printer out of paper or low on ink

If the printer goes out of paper or low on ink, then the printing operation is aborted and the necessary action needs to be taken, which can be feeding paper to the printer or replacing the ink cartridge. The use case ends.

6.6 Special Requirements

None

6.7 Related Use Cases

None

7. MAINTAIN LOGIN

7.1 Introduction

This use case documents the procedure for maintaining Login Details.

7.2 Actors

Librarian.

7.3 Pre-Condition

Librarian must be logged into the system.

7.4 Post-Condition

If use case is successful, the Login details should be updated, otherwise the system state is unchanged.

7.5 Flow of Events

7.5.1 Basic Flow

This use case starts when the Librarian wishes to add, delete or modify some details of login.

- The corresponding changes will be done.
- The use case ends.

7.5.2 Alternate Flow

None

7.6 Special Requirements

None

7.7 Related Use Cases

None

8. MAINTAIN STUDENT DETAILS

8.1 Introduction

This use case documents the procedure for maintaining student details.

8.2 Actors

Operator

8.3 Pre-Condition

Operator must be logged into the system.

8.4 Post-Condition

If use care is successful, the student details should be updated, otherwise the system state is unchanged.

8.5 Flow of Events

8.5.1 Basic Flow

This use case starts when the operator wishes to add, delete or modify some details of student.

- The corresponding changes will be done.
- The use case ends.

8.5.2 Alternate Flow

None

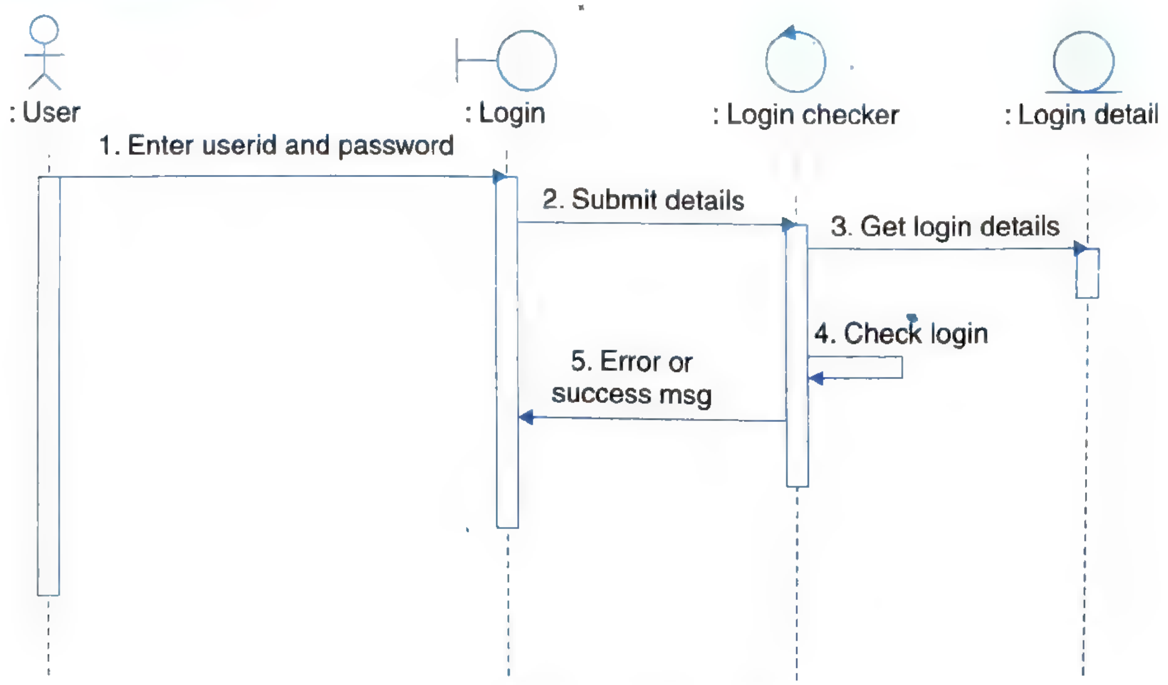
8.6 Special Requirements

None

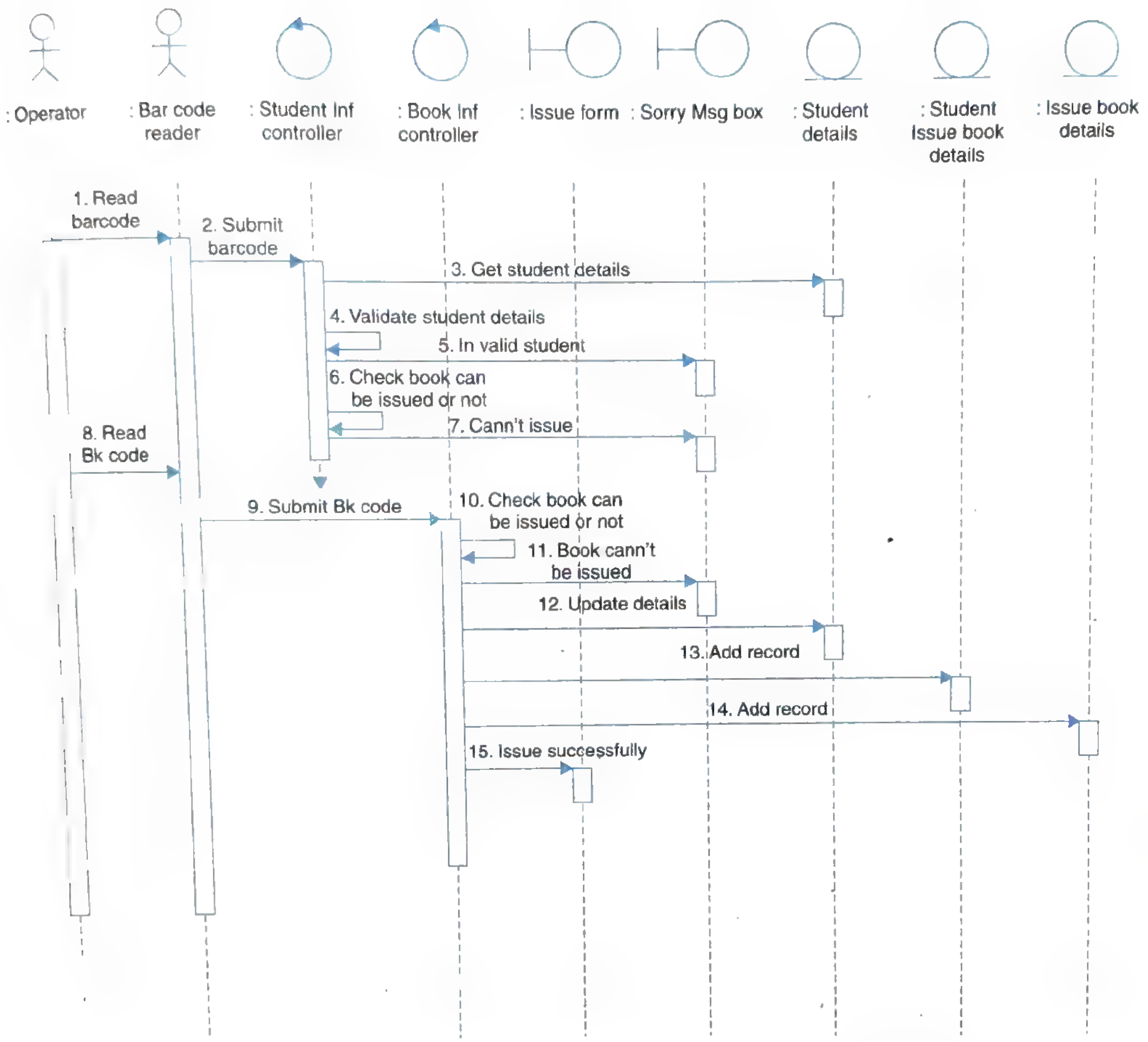
8.7 Related Use Cases

None

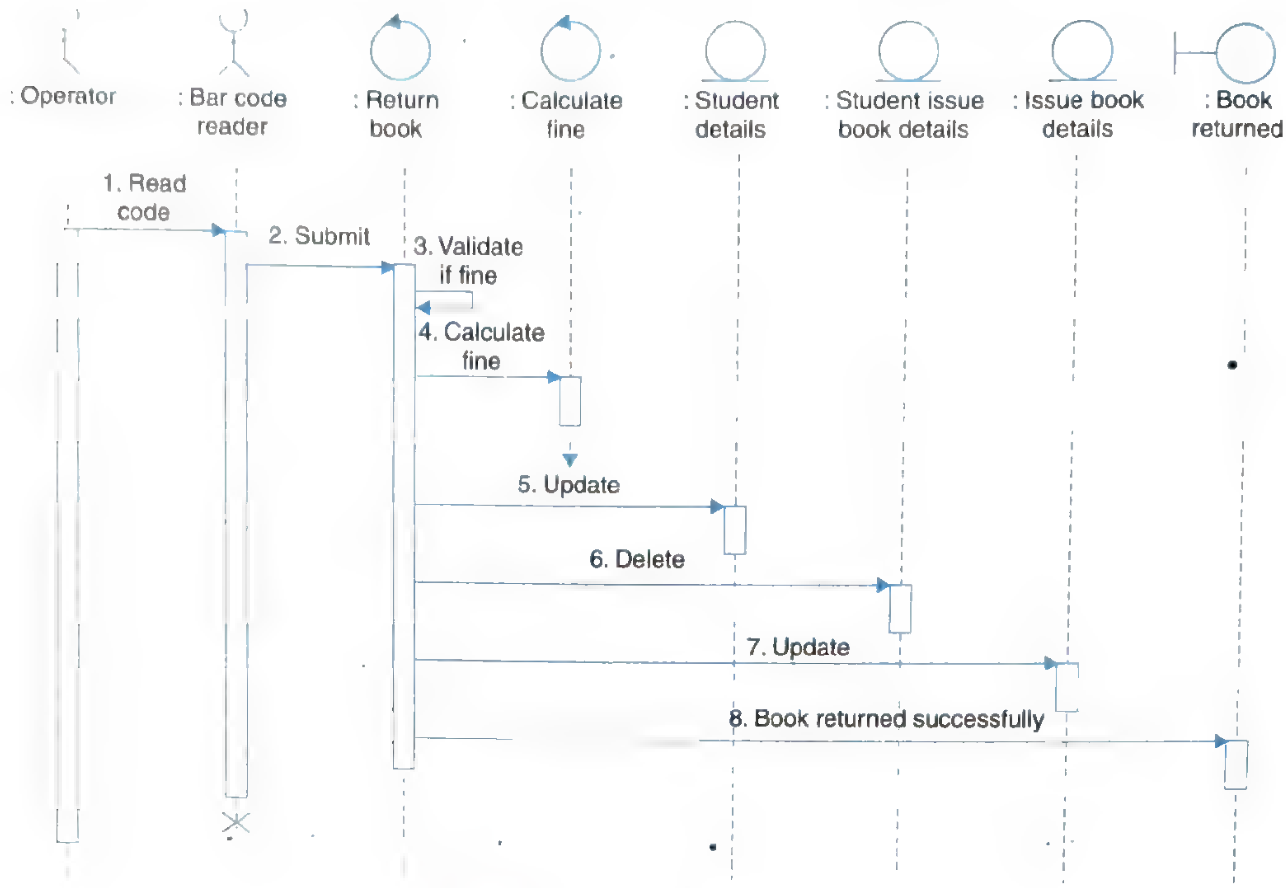
Sequence diagrams



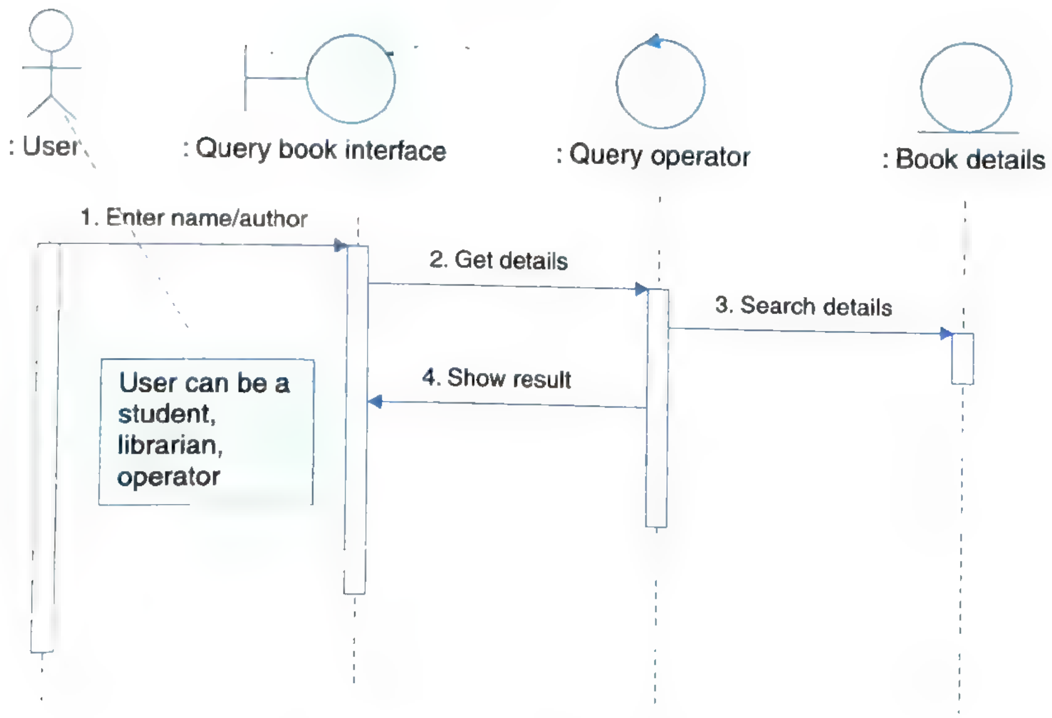
Sequence diagram—Login



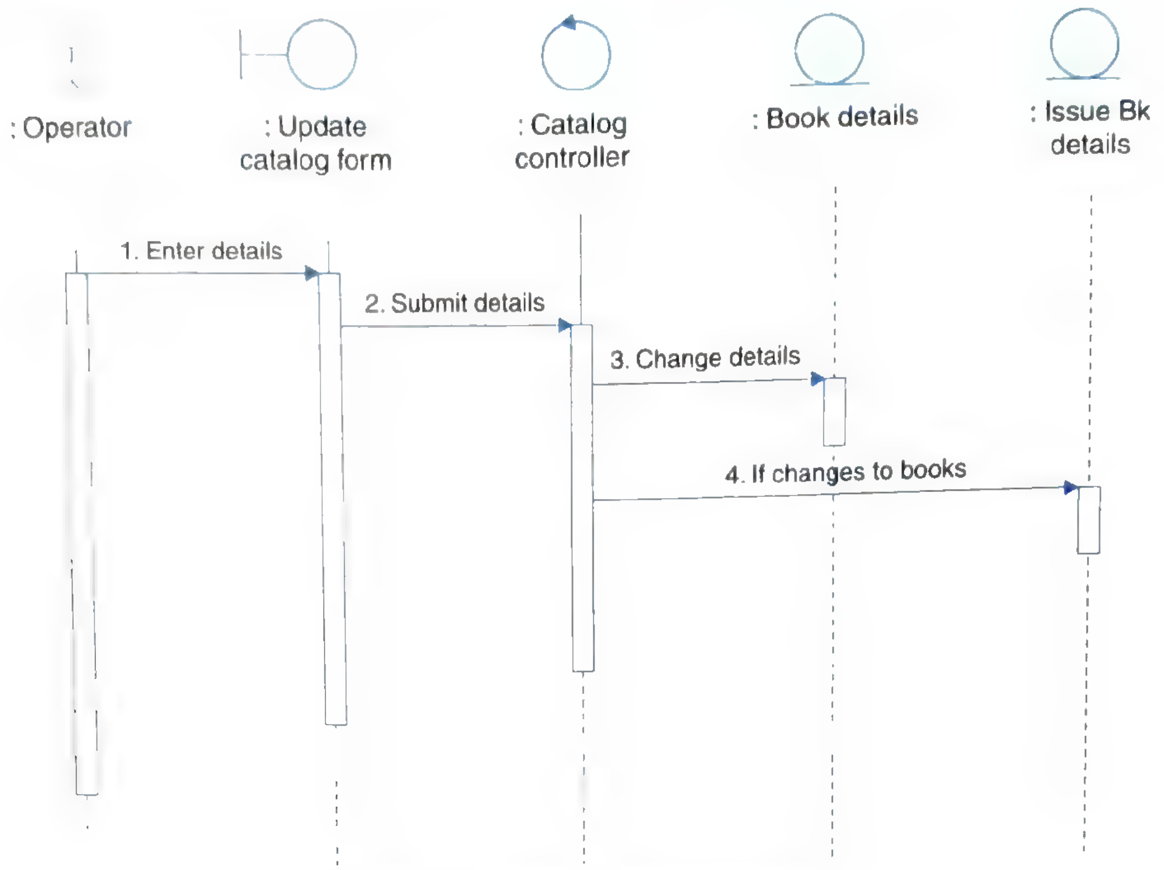
Sequence diagram—issue book



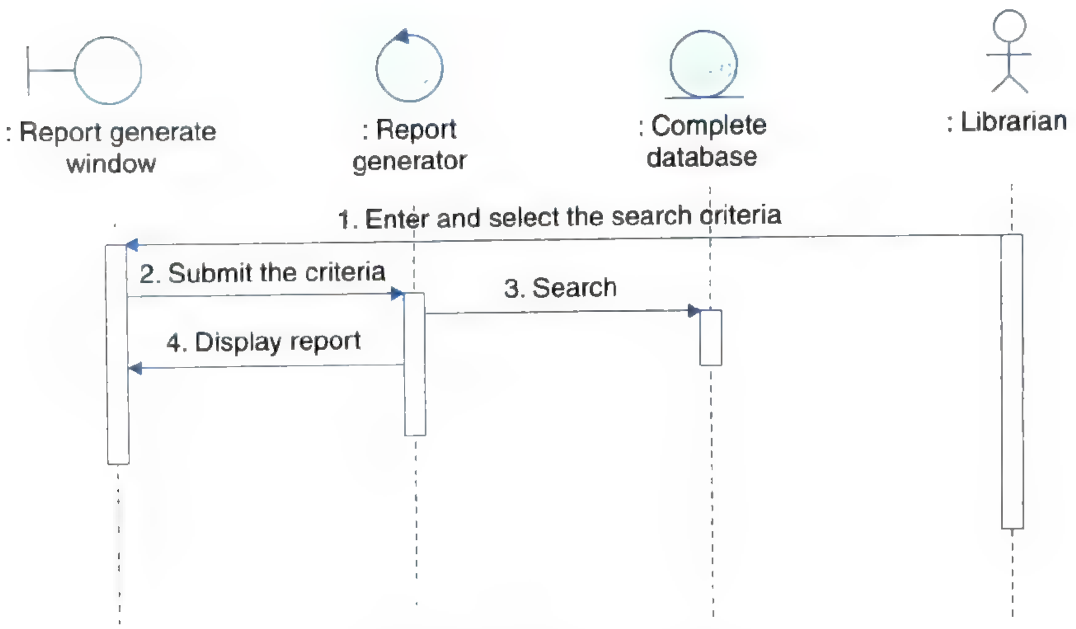
Sequence diagram—return book



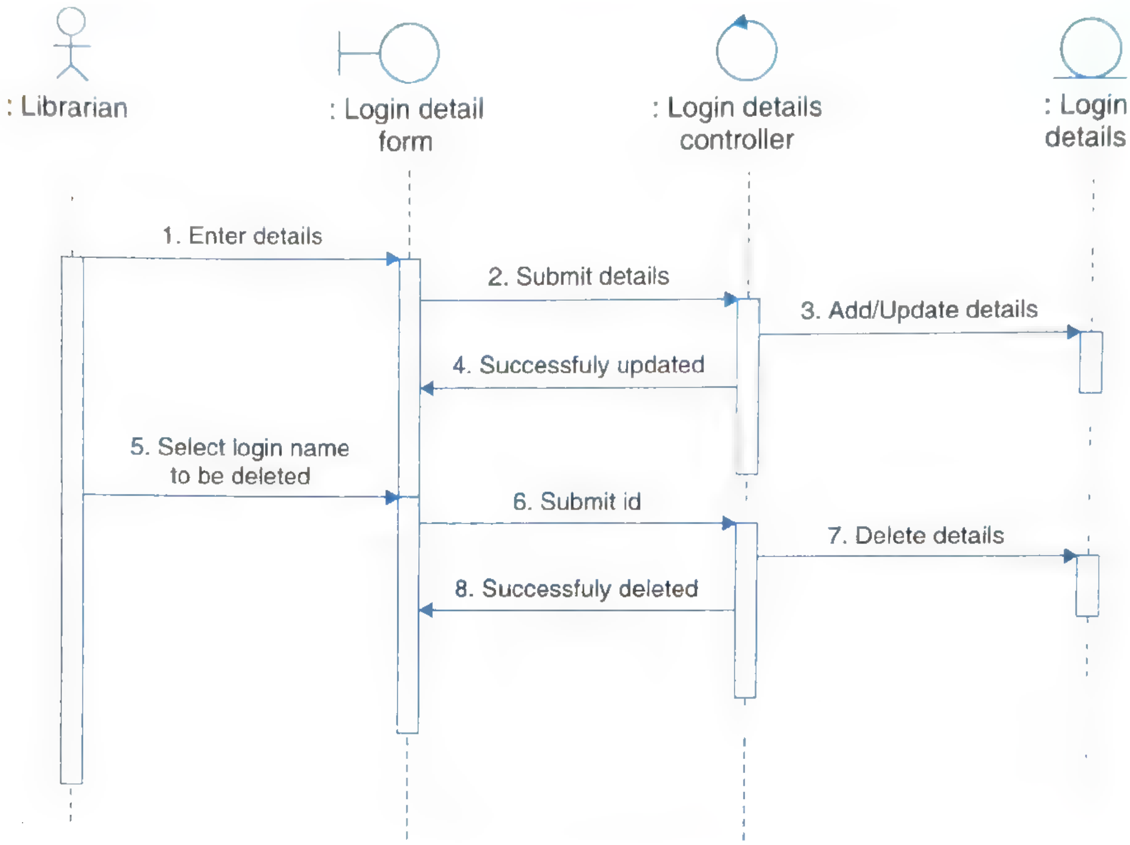
Sequence diagram—query book



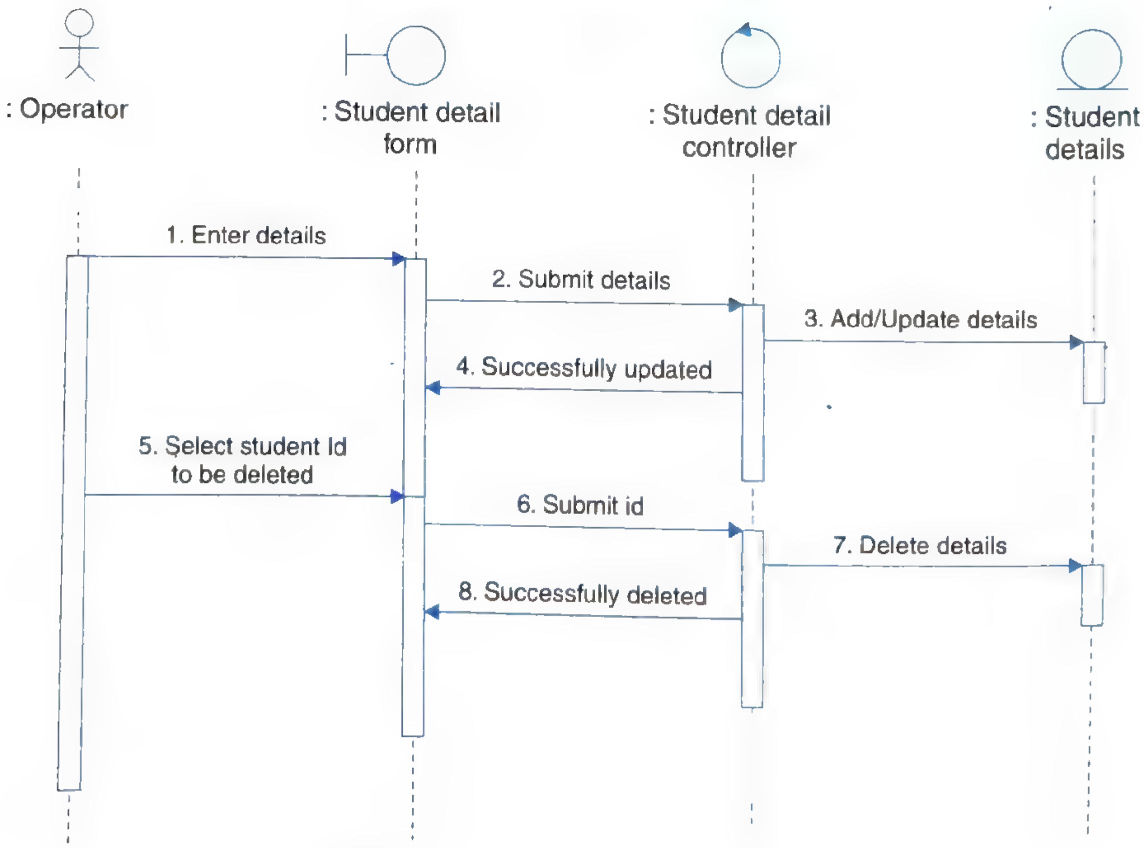
Sequence diagram—maintain catalog



Sequence diagram—generate reports

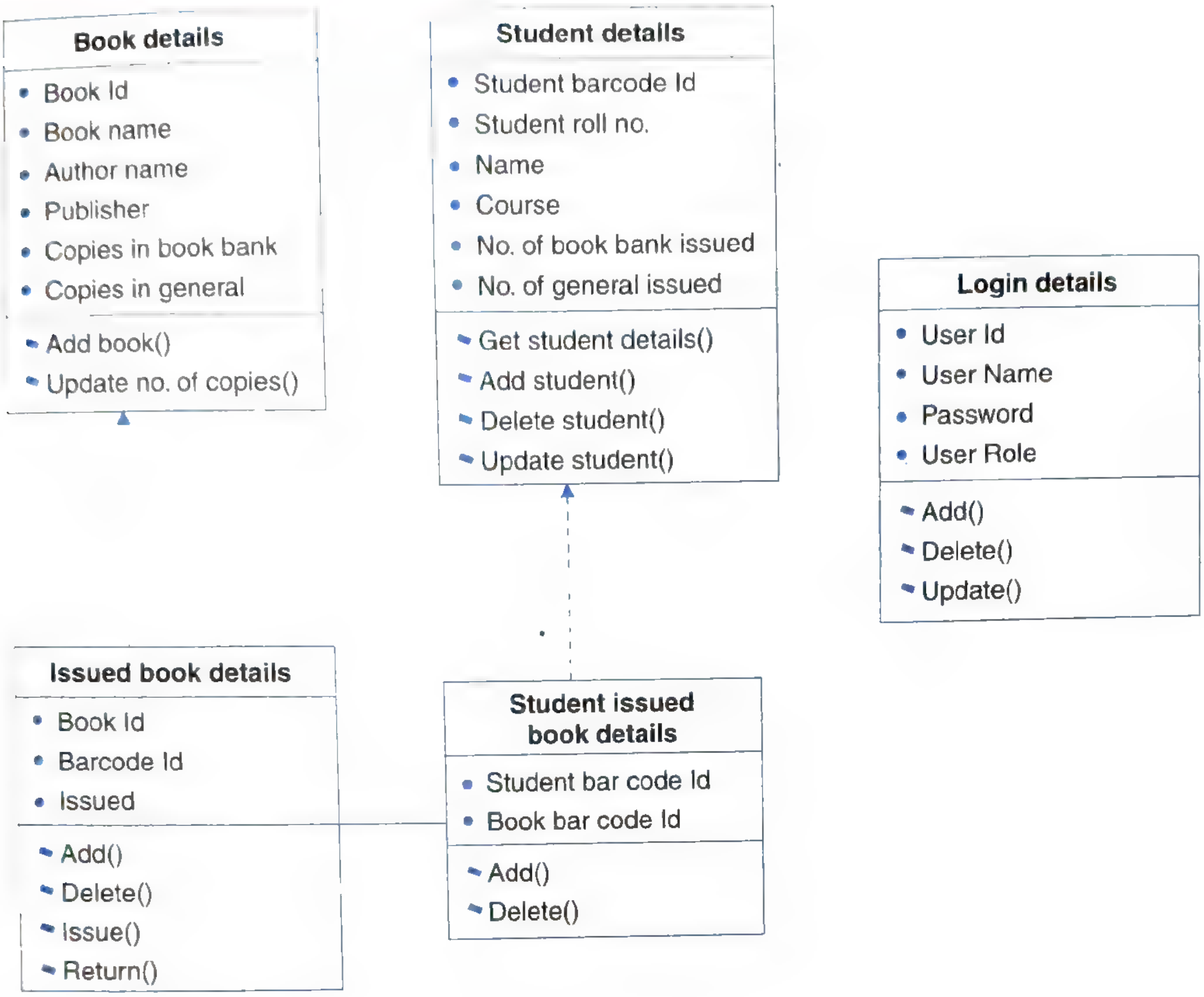


Sequence diagram—maintain login



Sequence diagram—maintain student details

Class diagram of entity classes



Class diagram of entity classes

In this design, we have shown class diagram of entity classes only. These entity classes generally becomes tables in the database. The other diagrams of the design document can be drawn using Rational Rose facilities (or other tools) and may or may not be required in each case study. All these diagrams are the foundations of the implementation phase.

REFERENCES

[KHAL01] Beth Khalil, "*Software Design and Development*", Homepage, http://attila.stevens-tech.edu, 2001.

[BOOC03] Booch G., "*Object Oriented Analysis and Design with Applications*" Pearson Edition, 2003

[FAIR2K] Fairly R., "*Software Engineering Concepts*", Tata McGraw Hill Pub., 2000.

[JACO98] Jacobson I., *et al.*, "*Object Oriented Software Engineering*", Addison Wesley 1998.

[JOSH03] Joshi S.D., *"Object Oriented Modelling and Design"*, Tech-Max, 2003.

[MYER78] Myers C., *"Composite Structured Design"*, Van Nostrand, Reinhold, 1978.

[FIEL01] Paul Field, *"An Introduction to Object Oriented Design"*, Paulfield @ dial.pipex.com, 2001.

[JALO98] Pankaj Jalote, *"An Integrated Approach to Software Engineering"*, Narosa Publications, Delhi, 1998.

[PFLE98] Pfleeger S.L., *"Software Engineering"*, Prentice-Hall International, Inc, 1998.

[SOMM2K] Sommerville I., *"Software Engineering"*, Addison-Wesley, 2000.

MULTIPLE CHOICE QUESTIONS

5.1. The most desirable form of coupling is
(*a*) control coupling (*b*) data coupling
(*c*) common coupling (*d*) content coupling.

5.2. The worst type of coupling is
(*a*) content coupling (*b*) common coupling
(*c*) external coupling (*d*) data coupling.

5.3. The most desirable form of cohesion is
(*a*) logical cohesion (*b*) procedural cohesion
(*c*) functional cohesion (*d*) temporal cohesion.

5.4. The worst type of cohesion is
(*a*) temporal cohesion (*b*) coincidental cohesion
(*c*) logical cohesion (*d*) sequential cohesion.

5.5. Which one is not a strategy for design ?
(*a*) bottom up design (*b*) top down design
(*c*) embedded design (*d*) hybrid design.

5.6. Temporal cohesion means
(*a*) cohesion between temporary variables (*b*) cohesion between local variables
(*c*) cohesion with respect to time (*d*) coincidental cohesion.

5.7. Functional cohesion means
(*a*) operations are part of single functional task and are placed in same procedures
(*b*) operations are part of single functional task and are placed in multiple procedures
(*c*) operations are part of multiple tasks
(*d*) none of the above.

5.8. When two modules refer to the same global data area , they are related as
(*a*) external coupled (*b*) data coupled
(*c*) content coupled (*d*) common coupled.

5.9. The module in which instructions are related through flow of control is
(*a*) temporal cohesion (*b*) logical cohesion
(*c*) procedural cohesion (*d*) functional cohesion.

5.10. The relationship of data elements in a module is called

(*a*) coupling (*b*) cohesion

(*c*) modularity (*d*) none of the above.

5.11. A system that does not interact with external environment is called

(*a*) closed system (*b*) logical system

(*c*) open system (*d*) hierarichal system.

5.12. The extent to which different modules are dependent upon each other is called

(*a*) coupling (*b*) cohesion

(*c*) modularity (*d*) stability.

EXERCISE

5.1. What is design ? Describe the difference between conceptual design and technical design.

5.2. Discuss the objectives of software design. How do we transform an informal design to a detailed design ?

5.3. Do we design software when we "write" a program ? What makes software design different from coding ?

5.4. What is modularity ? List the important properties of a modular system.

5.5. Define module coupling and explain different types of coupling.

5.6. Define module cohesion and explain different types of cohesion.

5.7. Discuss the objectives of modular software design. What are the effects of module coupling and cohesion ?

5.8. If a module has logical cohesion, what kind of coupling is this module likely to have with others ?

5.9. What problems are likely to arise if two modules have high coupling ?

5.10. What problems are likely to arise if a module has low cohesion ?

5.11. Describe the various strategies of design. Which design strategy is most popular and practical ?

5.12. If some existing modules are to be re-used in building a new system, which design strategy is used and why ?

5.13. What is the difference between a flow chart and a structure chart ?

5.14. Explain why it is important to use different notations to describe software designs.

5.15. List a few well-established function oriented software design techniques.

5.16. Define the following terms: Objects, Messages, Abstraction, Class, Inheritance and Polymorphism.

5.17. What is the relationship between abstract data types and classes ?

5.18. Can we have inheritance without polymorphism ? Explain.

5.19. Discuss the reasons for improvement using object-oriented design.

5.20. Explain the design guidelines that can be used to produce "good quality" classes or reusable classes.

5.21. List the points of a simplified design process.

5.22. Discuss the differences between object oriented and function oriented design.

5.23. What documents should be produced on completion of the design phase ?

5.24. Can a system ever be completely "decoupled" ? That is, can the degree of coupling be reduced so much that there is no coupling between modules ?

6 Software Metrics

An eminent physicist, Lord Kelvin said, "when you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science."

It is clear from Lord Kelvin's statement that everything should be measurable. If it is not measurable, we should make an effort to make it measurable. Thus, the area of measurement is very significant and important in all walks of life. When we discuss metrics for software engineering, situation is entirely different. Some feel that metrics are valuable management and engineering tools, while others feel that they are useless and expensive exercises in pointless data collection.

6.1 SOFTWARE METRICS: WHAT AND WHY ?

Science begins with quantification; we cannot do physics without a notion of length and time; we cannot do thermodynamics until we measure temperature. All engineering disciplines have metrics (such as metrics for weight, density, wave length, pressure and temperature) to quantify various characteristics of their products. The most fundamental question we can ask is "how big is the program"? Without defining what big means, it is obvious that it makes no sense to say, "this program will need more testing than that program" unless we know "how big they are relative to one another. Comparing two strategies also needs a notion of size. The number of tests required by a strategy should be normalized to size. For example A needs 1.4 tests per unit of size, while strategy B needs 4.3 tests per unit of size.

What is meant by size was not obvious in the early phases of science development. Newton's use of mass instead of weight was a breakthrough for physics, and early researchers in thermodynamics had heat, temperature, and entropy hopelessly confused. Size is not obvious for the software. Metrics must be objective in the sense that the measurement process is algorithmic and will yield the same results no matter who applies it [BEIZ90]. To see what kinds of metrics, we need, let us ask some questions.

1. How to measure the size of a software?
2. How much will it cost to develop a software?
3. How many bugs can we expect?
4. When can we stop testing?
5. When can we release the software?

6. What is the complexity of a module?
7. What is the module strength and coupling?
8. What is the reliability at the time of release?
9. Which test technique is more effective?
10. Are we testing hard or are we testing smart?
11. Do we have a strong program or a weak test suite?

If we want an answer to the above questions, we will have to do our own measuring and fit our own empirical laws to the measured data. Most of the metrics are aimed at getting empirical laws that relate program size (however it be measured) to expected number of bugs, expected number of tests required to find bugs, test technique effectiveness, resource requirement, release instant, reliability and quality requirement, etc.

The groundwork of software metrics was laid in the seventies. The earliest paper on the subject was of course published in 1968 [RUBE68]. From these earlier works, interesting results have emerged in the eighties. The term software metrics designates here "a unit of measurement of a software product or software related process [HAME85]."

The terms measure, measurement and metrics are often used interchangeably. However, the difference amongst these should be understood clearly. Pressman explained [PRESO5] as : "A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute". A measure may be number of errors found in a module during testing. Measurement is the result of such data collection. A software metric should relate the individual measures in some way like : average number of errors found per hour of testing.

We want to measure various characteristics of software. Some are direct measures and others are indirect measures. Fenton [FENTO4] defined measurement as :

"It is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules."

Here, we want to collect information about attributes of entities. An entity is an object (such as car, hospital) or an event (such as journey or surgical operation in hospital) in the real world. An attribute is a feature or property of an entity. Examples are colour of a car, type of hospital, cost of journey, seriousness of the operation etc.

Some of the direct measures of software process are time and effort required, maturity of the process etc. Some of the direct measures of software product are lines of code produced, number of defects found, execution speed etc. Indirect measures of software product may include complexity, efficiency, reliability, portability, maitainability etc.

6.1.1 Definition

Software metrics can be defined as [GOOD93] *"The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products".*

This definition covers quite a lot. Software metrics are all about measurements which, in turn, involve numbers, the use of numbers to make things better, to improve the process of developing software and to improve all aspects of the management of that process. Software metrics are applicable to the whole development life cycle from initiation, when costs must be estimated, to monitoring the reliability of the end product in the field, and the way that product changes over time with enhancement. It covers the techniques for monitoring and controlling the progress of the software development, such that the fact that it is going to be six months late is recognized as early as possible rather than the day before delivery is due. It even covers organizations determining which of its software products are the cash cows and which are the dogs.

6.1.2 Areas of Applications

The most established area of software metrics is cost and size estimation techniques. There are many proprietary packages in the market that provide estimates of software system size, cost to develop a system, and the duration of the development or enhancement of the project. These packages are based on estimation models, like COCOMO81, COCOMO-II, developed by Berry Boehm [BOEH81]. Various techniques that do not require the use of readymade tools are also available. There has been a great deal of research carried out in this area, and this research continues in all important software industries and other organizations. One thing that does come across strongly from the results of this research work is that organizations cannot rely, solely, on the use of proprietary packages.

Controlling software development projects through measurement is an area that is generating a great deal of interest. This has become much more relevant with the increase in fixed price contracts and the use of penalty clauses by customers who deal with software developers.

The prediction of quality levels for software, often in terms of reliability, is another area where software metrics have an important role to play. Again, there are proprietary models in the market that can assist this, but debate continues about their accuracy. The requirement is there both from the customer's point of view and that from the developer's, who needs to control testing and other costs. Various techniques can be used now, and this area will become more and more important in future.

The use of software metrics to provide quantitative checks on software design is also a well-established area. Much research has been carried out, and some organizations have used such techniques to very good effect. This area of software metrics is also being used to control software products, which are in place and are subject to enhancement.

Software metrics are also used to provide management information. This includes information about productivity, quality and process effectiveness. It is important to realize that this should be seen as an ongoing activity. Snapshots of the current situation have their place, but the most valuable information comes when we can see trends in data. Is productivity or quality getting better or worse over time? Why is this happening? What can management do to improve things? The provision of management information is as much an art as a science. Statistical analysis is a part of it, but the information must be presented in a way that managers find it useful, at the right time and for the right reasons. All this shows that software metrics is a vast field and have wide variety of applications throughout the software life cycle [GOOD93].

6.1.3 Problems During Implementation

Implementing software metrics in an organization of any size is difficult. There are many problems that have to be overcome and decisions that have to be made, often with limited information on hand. The first decision concerns the scope of the work. There is a rule in software development that we do not try something new on a large or critical system and this translates, in the software metrics area, to *'do not try to do too much'*. On the other hand, there is evidence that concentrating on too small an area can result in such a limited pay back as to invalidate software metrics in an organization. It is always said, *'do not bet your career on a single metric'*.

Another problem during implementation arises if we start measuring the performance of individuals. There was an organization, which used individual productivity, in terms of functionality divided by effort, as a major determinant of salary increase. While this may appear attractive to some managers, the organization later stated that this was one of the worst mistakes it ever made. Using measurement in this way is counter productive, divisive and simply ensures that developers will rig the data they supply. We may say that management has one chance and one chance only. The first time that a manager uses data supplied by an individual against that individual is the last time that the manager will get accurate or true data from that person. The reasoning behind such a statement is simple, *"employees do not like upsetting the boss!"*

There is a great temptation to seek the "silver bullet", the single measure that tells all about the software development process. Currently, this does not exist. We must realize that implementing software metrics means changing the way in which people work and think. The software engineering industry is maturing. Customers are no longer willing to accept poor quality and late deliveries. Management is no longer willing to pour money into the black hole of IT, and more management control is now a key business requirement. Competition is growing. Developers have to change and mature as well, and this can be a painful experience as they find tenets of their beliefs being challenged and destroyed. Some simple statements that could be made by many in our industry together with interpretation of current management trends will illustrate the following points [GOOD93].

- **Statement** : Software development is so complex; it cannot be managed like other parts of the organization.

  Management view: Forget it, we will find developers and managers who will manage that development.
- **Statement** : I am only six months late with this project.

  Management view: Fine, you are only out of a job.
- **Statement** : But you cannot put reliability constraints in the contract.

  Management view: Then we may not get the contract.

The list is almost endless and the message is clear.

Software metrics cannot solve all our problems, but they can enable managers to improve their processes, to improve productivity and quality, to improve the probability of survival. But it is not an easy option. Many metrics programs fail. One reason for this is that organizations and individuals who would never dream of introducing a new system without a

structured approach ignore the problems of introducing change inherent in software metrics implementation. Only by treating the implementation of software metrics as a project or programme in its own right with plans, budgets, resources and management commitment can make such an implementation succeed. Hence the use of software metrics does not ensure survival, but it improves the probability of survival.

6.1.4 Categories of Metrics

There are three categories of software metrics which are given below:

(*i*) **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

(*ii*) **Process metrics:** describe the effectiveness and quality of the processes that produce the software product. Examples are:

- effort required in the process
- time to produce the product
- effectiveness of defect removal during development
- number of defects found during testing
- maturity of the process.

(*iii*) **Project metrics:** describe the project characteristics and execution. Examples are:

- number of software developers
- staffing pattern over the life cycle of the software
- cost and schedule
- productivity

Some metrics belong to multiple categories like quality metric may belong to all three categories. It focuses on the quality aspects of the product process, and the project. Some important metrics are discussed in subsequent sections of the chapter.

6.2 TOKEN COUNT

Two important size metrics (LOC and Function count) are discussed in chapter 4 (software project planning). These metrics have established their applicability in the field, specifically as a key input in the costing models like COCOMO, COCOMO-II, Putnam resource allocation model etc. The major problem with LOC measure is that it is not consistent because some lines are more difficult to code than others. A program is considered to be a series of tokens and if we count the number of tokens, some interesting results may emerge.

In the early 1970s, the late Professor Maurice Halstead and his co-workers at Purdue University developed the software science family of measures [HALS77]. Tokens are classified as either operators or operands. All software science measures are functions of the counts of these tokens.

Generally, any symbol or keyword in a program that specifies an algorithmic action is considered an operator, while a symbol used to represent data is considered an operand. Most punctuation marks are also categorized as operators. Variables, constants and even labels are operands. Operators consist of arithmetic symbols such as +, –, /, * and command names such

as "while", "for", "printf", special symbols such as :=, braces, parentheses, and even function names such as "eof" (end of file). The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

$$\eta = \eta_1 + \eta_2 \tag{6.1}$$

where η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

The length of the program in terms of the total number of tokens used is

$$N = N_1 + N_2 \tag{6.2}$$

where N : program length.

N_1 : total occurrences of operators

N_2 : total occurrences of operands

It should be noted that N is closely related to the lines of code (LOC) measure of program. For machine language programs where each line consists of one operator and one operand, the program length is

$$N = 2 * LOC \tag{6.3}$$

Additional metrics are defined using these basic terms. Another measure for size of the program is called the volume.

$$V = N * \log_2\eta \tag{6.4}$$

The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used. Volume may also be interpreted as the number of mental comparisons needed to write a program of length N, assuming a binary search method is used to select a member of the vocabulary of size η. Since an algorithm may be implemented by many different but equivalent programs, a program that is minimal in size is said to have the potential volume V*. Any given program with volume V is considered to implement at the program level L, which is defined by

$$L = V^* / V \tag{6.5}$$

The value of L ranges between zero and one, with L = 1 representing a program written at the highest possible level (*i.e.*, with minimum size). The inverse of the program level is termed the difficulty. That is

$$D = 1 / L \tag{6.6}$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

The effort required to implement a program increases as the size of the program increases. It also takes more effort to implement a program at a lower level (higher difficulty) when

compared with another equivalent program at a higher level (lower difficulty). Thus the effort in software science is defined as

$$E = V / L = D * V \tag{6.7}$$

The unit of measurement of E is elementary mental discriminations.

6.2.1 Estimated Program Length

As stated earlier, size of the program is the total number of tokens, referred to as program length, N. The first hypothesis of software science is that the length of a well-structured program is a function only of the number of unique operators and operands. This function, called the estimated length equation is denoted by $\hat{N}$ and is defined by

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{6.8}$$

For example, the sorting program in Fig. 4.2 (given in chapter 4) has 14 unique operators and 10 unique operands. Suppose that these numbers are known before the completion of the program, possibly by using a program design language and knowing the programming language chosen. It is then possible to estimate the length N of the program in number of tokens using equation 6.8.

$$\hat{N} = 14 \log_2 14 + 10 \log_2 10$$
$$= 53.34 + 33.22 = 86.56$$

Thus, even before constructing the sorting program, equation 6.8 predicts that there will be about 87 tokens (operators and operands) used in the completed program, which is within 10% of the program's actual length N of 91 tokens (refer Table 6.2).

Refer Variables

There are several major flaws in the derivation of the length equation (equation 6.8). These flaws are discussed in [SHEN83]. Most of empirical support for software science is based on analysis of the relationship between estimated and actual lengths. Researchers frequently report correlation of 0.95 or higher between these quantities [FITZ78, CONT86].

The length equation can be viewed as a hypothesis whose credibility rests on several independent experiments, which have indicated that $\hat{N}$ is a reasonable estimator of N. In the process of studying Halstead's length estimator the following alternate expressions have been published to estimate program length.

$$N_J = \log_2 (\eta_1!) + \log_2 (\eta_2!) \tag{6.9}$$

This equation was proposed by Jensen & Variavan [JENS85] and found to be a much better approximation for their data set. Another equation was derived by Mehndiratta and Grover [MEHN86], which is the slight modification of Halstead length estimator.

$$N_B = \eta_1 \log_2 \eta_2 + \eta_2 \log_2 \eta_1 \tag{6.10}$$

The Halstead estimator was further modified by Card and Agresti [CARD87] by substituting $\sqrt{\eta_1}$ for $\log_2 \eta_1$ and $\sqrt{\eta_2}$ for $\log_2 \eta_2$, thus

$$N_C = \eta_1 \sqrt{\eta_1} + \eta_2 \sqrt{\eta_2} \tag{6.11}$$

It is claimed that for small values of η, $\sqrt{\eta}$ and $\log_2 \eta$ behave similarly. Unfortunately, the accuracy of the estimator does not improve.

Although it is easy to construct a pathological program to make $\hat{N}$ a poor predictor of N, there is overwhelming evidence using existing analysers to suggest the validity of the length equation in several languages. Shen et.al. [SHEN83] suggested that a misclassification of any token has virtually no effect on the final estimate.

Since $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \approx (\eta \log_2 \eta) / 2$ regardless of how the vocabulary of size η is divided into operators and operands. Hence

$$N_S = (\eta \log_2 \eta) / 2 \tag{6.12}$$

The definitions of unique operators, unique operands, total operators and total operands are not specifically delineated. Bulut [BULU73] has considered the counting methods used for FORTRAN, ALGOL & machine languages. Conte [CONE86] suggested counting rules for PASCAL and James Elshoff [JAME78] reported rules for PL/1 programs. Counting rules for C languages are suggested [YOGE95] and are given below:

Counting rules for C language

1. Comments are not considered.
2. The identifier and function declarations are not considered.
3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.
5. Local variables with the same name in different functions are counted as unique operands.
6. Function calls are considered as operators.
7. All looping statements *e.g.*, do{...} while ( ), while ( ) {...}, for ( ) {...}, all control statements *e.g.*, if ( ) {...}, if ( ) {...} else {...}, etc. are considered as operators.
8. In control construct switch ( ) { case : ...}, switch as well as all the case statements are considered as operators.
9. The reserve words like return, default, continue, break, size of, etc., are considered as operators.
10. All the brackets, commas, and terminators are considered as operators.
11. GOTO is counted as an operator and the label is counted as an operand.
12. The unary and binary occurrence of "+" and "–" are dealt separately. Similarly "*" (multiplication operator), and "*" (de referencing operator), "&" (bitwise AND operator) and "&" (address operator) are dealt with separately.
13. In the array variables such as "array-name[index]" "array-name" and "index" are considered as operands and [ ] is considered as operator.
14. In the structure variables such as "struct-name, member-name" or "struct-name →member-name", struct-name, member-name are taken as operands and '.', '→' are taken as operators. Same names of member elements in different structure variables are counted as unique operands.
15. All the hash directives are ignored.

The counting method presented here is fairly complete, however, "C" is so rich and complex that probability of additions and exceptions is always there.

6.2.2 Potential Volume

Many different but equivalent programs may implement an algorithm. Amongst all these programs, the one that has minimal size is said to have the potential volume V*. Halstead argued that the minimal implementation of any algorithm was through a reference to a procedure that had been previously written. The implementation of this algorithm would then require nothing more than invoking the procedure and supplying the operands for its input and output parameters. It is shown that the potential volume of an algorithm implemented as a procedure call could be expressed as

$$V^* = (2+\eta_2^*)\log_2(2+\eta_2^*) \tag{6.13}$$

The first term in the parentheses, 2, represents the two unique operators for the procedure call — the procedure name and a grouping symbol that separates the procedure name from its parameters. The second term, η_2^* represents the number of conceptually unique input and output parameters. η_2^* can probably be determined for small application programs, it is much more difficult to compute for large programs, such as compiler or an operating system. In these cases it is difficult to identify precisely the conceptually unique operands.

6.2.3 Estimated Program Level / Difficulty

Halstead offered an alternate formula that estimates the program level.

$$\hat{L} = 2\,\eta_2 / (\eta_1 N_2) \tag{6.14}$$

Hence $$\hat{D} = \frac{1}{\hat{L}} = \frac{\eta_1 N_2}{2\eta_2}$$

An intuitive argument for this formula is that programming difficulty increases, if additional operators are introduced (*i.e.*, if $\eta_1/2$ increases) and if an operand is used repetitively (*i.e.*, if N_2/η_2 increases) . Every parameter in equation 6.14 may be obtained by counting the operators and operands in a finished computer program.

6.2.4 Effort and Time

Halstead hypothesized that the effort required to implement a program increases as the size of the program increases. It also takes more effort to implement a program at a lower level (higher difficulty) than another equivalent program at a higher level (lower difficulty). Thus, effort E measured in elementary mental discriminations is:

$$\begin{aligned} E &= V / \hat{L} = V * \hat{D} \\ &= (\eta_1 N_2 N \log_2 \eta) / 2\eta_2 \end{aligned} \tag{6.15}$$

A major claim for software science is its ability to relate its basic metrics to actual implementation. A psychologist, John Stroud, suggested that the human mind is capable of making a limited number of elementary discriminations per second [STRO67]. Stroud claimed that this number β (called the stroud number) ranges between 5 and 20. Since effort E uses elementary mental discriminations as its unit of measure, the programming time T of a program in seconds is simply

$$T = E / \beta \qquad (6.16)$$

β is normally set to 18 since this seemed to give best results in Halstead's earliest experiments, which compared the predicted times with observed programming times, including the time for design, coding, and testing. Halstead claimed that this formula can be used to estimate programming time when a problem is solved by one proficient, concentrating programmer writing a single module program.

6.2.5 Language Level

There are now literally hundreds of programming languages. In some organizations, several languages may be used on a regular basis for software development. For example, in some large companies, software is being developed using FORTRAN, PASCAL, COBOL, and assembly language. There are proponents of each major language (including those of Ada, C, and Prolog) that argue that their favourite language is the best to use. These arguments suggest the need for a metric that expresses the power of a language [SHEN83].

Halstead hypothesized that, if the programming language is kept fixed, as V^* increases L decreases in such a way that the product $L \times V^*$, remains constant. Thus, this product, which he called the language level, λ can be used to characterize a programming language. Lower value of λ means that the language is closer to the machine.

$$\lambda = L \times V^* = L^2V \qquad (6.17)$$

Using this formula, Halstead and other researchers determined the language level for various languages as shown in Table 6.1 [HALS77, SHEN83, YOGE95].

Table 6.1: Language levels

Language	*Language level λ*	*Variance σ*
PL/1	1.53	0.92
ALGOL	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42
PASCAL	2.54	–
APL	2.42	–
C	0.857	0.445

These averages, λ's follow the intuitive rankings of most programmers for these languages, but they all have large variances.

The current state of software science seems to be still that of a evolving theory. There are those who question (with good reason in most cases) some of its underlying assumptions. However, there is large body of published data that suggests that software science metrics may be useful; especially η_1 and η_2 have been shown to strongly correlate to program size and error rates. Researchers are therefore continuing to find these metrics useful as a basis for size and effort models. Thus, Halstead work served to stimulate a great deal of interest in software metrics, as well as to contribute some basic metrics that survive till today.

Example 6.1

Consider the sorting program given in Fig. 4.2 of chapter 4 (software project planning). List out the operators and operands and also calculate the values of software science measures like η, N, V, E, λ etc.

Solution

The list of operators and operands is given in Table 6.2.

Table 6.2: Operators and operands of sorting program of Fig. 4.2 of chapter 4.

Operators	*Occurrences*	*Operands*	*Occurrences*
int	4	SORT	1
( )	5	*x*	7
,	4	*n*	3
[ ]	7	*i*	8
if	2	*j*	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
< =	2	—	—
+ +	2	—	—
return	2	—	—
{ }	3	—	—
$\eta_1 = 14$	$N_1 = 53$	$\eta_2 = 10$	$N_2 = 38$

Here $N_1 = 53$ and $N_2 = 38$. The program length $N = N_1 + N_2 = 91$

Vocabulary of the program $\eta = \eta_1 + \eta_2 = 14 + 10 = 24$

Volume $V = N \times \log_2 \eta$

$= 91 \times \log_2 24 = 417$ bits.

If a binary encoded scheme is used to represent each of the 24 items in the vocabulary, it would take 5 bits per item, since a 4 bit scheme leads to 16 unique codes (which is not enough), and a 5-bit scheme leads to 32 unique codes (which is more than sufficient). Each of the 91 tokens used in the program could be represented in order by a 5-bit code, leading to a string of 5 × 91 = 455 bits that would allow us to store the entire program in memory. Notice that size analysis is based on storing not a compiled version of the subroutine, but a binary translation of the original program. We could then say that this program occupies 455 bits of storage in its encoded form. However, also notice that a 5-bit scheme allows for 32 tokens

instead of just 24 tokens, so instead of using the integer 5, volume uses the non-integer $\log_2 \eta = 4.58$ to arrive at a slightly smaller volume of 417.

The estimated program length $\hat{N}$ of the program

$$= 14 \log_2 14 + 10 \log_2 10$$
$$= 14 * 3.81 + 10 * 3.32$$
$$= 53.34 + 33.2 = 86.45$$

Conceptually unique input and output parameters are represented by η_2^*

$\eta_2^* = 3$ {x : array holding the integer to be sorted. This is used both as input and output}.

{N : the size of the array to be sorted}.

The potential volume $V^* = 5 \log_2 5 = 11.6$

Since $L = V^* / V$

$$= \frac{11.6}{417} = 0.027$$

$$D = I / L$$

$$= \frac{1}{0.027} = 37.03$$

Estimated program level

$$\hat{L} = \frac{2}{\eta_1} \times \frac{\eta_2}{N_2} = \frac{2}{14} \times \frac{10}{38} = 0.038$$

which is not very close to the 0.027 level, we determined earlier using the conceptually unique operands.

We may use another formula

$$\hat{V}^* = V \times \hat{L} = 417 \times 0.038 = 15.67$$

The discrepancy between V^* and $\hat{V}^*$ does not inspire confidence in the application of this portion of software science theory to more complicated programs.

$$E = V/\hat{L} = \hat{D} \times V$$
$$= 417 / 0.038 = 10973.68$$

Therefore, 10974 elementary mental discriminations are required to construct the program.

$$T = E / \beta = \frac{10974}{18} = 610 \text{ seconds} \approx 10 \text{ minutes}$$

This is probably a reasonable time to produce the program, which is very simple.

Table 6.3

```
#include < stdio.h >
#define MAXLINE 100
int getline(char line[],int max);
int strindex(char source[],char search for[]);
char pattern[ ]=“ould”;
int main()
{
    char line[MAXLINE];
    int found = 0;
    while(getline(line,MAXLINE)>0)
        if(strindex(line, pattern)>=0)
        {
            printf(“%s”,line);
            found++;
        }
    return found;
}
int getline(char s[],int lim)
{
    int c,i=0;
    while(--lim > 0 && (c=getchar())!= EOF && c!=‘\n’)
        s[i++]=c;
    if(c==‘\n’)
        s[i++] = c;
    s[i] = ‘\0’;
    return i;
}
int strindex(char s[],char t[])
{
```

```
    int i,j,k;
    for(i=0;s[i] !='\0';i++)
    {
        for(j=i,k=0;t[k] != '\0',s[j] ==t[k];j++,k++);
        if(k>0 && t[k] =='\0')
            return i;
    }
    return -1;
}
```

Example 6.2

Consider the program shown in Table 6.3. Calculate the various software science metrics.

Solution

List of operators and operands are given in Table 6.4.

Table 6.4

Operators	*Occurrences*	*Operands*	*Occurrences*
main ()	1	—	—
–	1	**Extern variable** pattern	1
for	2	**Main function** line	3
= =	3	found	2
! =	4	**Getline function** s	3
getchar	1	lim	1
( )	1	*c*	5
&&	3	*i*	4
– –	1	**Strindex function** *s*	2
return	4	*t*	3
+ +	6	*i*	5
printf	1	*j*	3
> =	1	*k*	6
strindex	1	**Numerical operands** 1	1
If	3	Maxline	1
>	3	0	8
getline	1	'\0'	4

(Contd.)...

while	2	'\n'	2
{ }	5	**Strings "ould"**	1
=	10	—	—
[ ]	9	—	—
.	6	—	—
;	14	—	—
EOF	1	—	—
$\eta_1 = 24$	$N_1 = 84$	$\eta_2 = 18$	$N_2 = 55$

Program vocabulary $\eta = 42$

Program length $N = N_1 + N_2$

$= 84 + 55 = 139$

Estimated length $\hat{N} = 24 \log_2 24 + 18 \log_2 18 = 185.115$

% error $= 24.91$

Program volume $V = 749.605$ bits

Estimated program level $= \frac{2}{\eta_1} \times \frac{\eta_2}{N_2}$

$$= \frac{2}{24} \times \frac{18}{55} = 0.02727$$

Minimal volume $V^* = 20.4417$

Effort $= V/\hat{L}$

$$= \frac{749.605}{.02727}$$

= 27488.33 elementary mental discriminations.

$$\text{Time T} = E/\beta = \frac{27488.33}{18}$$

= 1527.1295 seconds

= 25.452 minutes

6.3 DATA STRUCTURE METRICS

Essentially, the need for software development and other activities are to process data. Some data is input to a system, program, or module; some data may be used only internally; and some data is the output from a system, program, or module. A few examples of input, internal, and output data appear in Fig. 6.1.

Program	*Data input*	*Internal data*	*Data output*
Payroll	Name / Social Security No. / Pay Rate / Number of hours worked	Withholding rates Overtime factors Insurance premium Rates	Gross pay withholding Net pay Pay ledgers
Spread sheet	Item Names / Item amounts / Relationships among items	Cell computations Sub-totals	Spreadsheet of items and totals
Software Planner	Program size / No. of software developers on team	Model parameters Constants Coefficients	Est. project effort Est. project duration

Fig. 6.1: Some examples of input, internal, and output data [CONT86]

Thus, an important set of metrics is that, which captures the amount of data input to, processed in and output from software. For example, assume that a problem can be solved in two ways, resulting in programs A and B. A has 25 input parameters, 35 internal data items, and 10 output parameters. B has 5 input parameters, 12 internal data items, and 4 output parameters. We can assume that A is probably more complicated, took more time to program, and has a greater probability of errors than B.

A count of the amount of data input to, processed in, and output from software is called a data structure metric. This section presents several data structure metrics. Some concentrate on variables (and even constants) within each module and ignore the input/output dependencies. Others concern themselves primarily with the input/output situation. There is no general agreement on how the line of code measure (the classical and best-known software metric) is to be counted. Thus, it is not surprising that there are various methods for measuring data structures as well. In the following subsections, we will discuss the metrics proposed to measure the amount of data, the usage of data within modules, and the degree to which data is shared among modules [CONT86].

6.3.1 The Amount of Data

Most compilers and assemblers have an option to generate a cross-reference list, indicating the line where a certain variable is declared and the line or lines where it is referenced. Such a list is useful in debugging and maintenance, and can help determine the amount of data in the program. Consider a simple program which appears in Fig. 6.2. It inputs work hours and pay rates, and computes gross pay, taxes, and net pay. The C compiler produces a cross-reference listing for this program, which appears in Fig. 6.3.

One method for determining the amount of data is to count the number of entries in the cross-reference list. Be careful to exclude from the count those variables that are defined but never used. The definitions of these variables may be made for future reference, but they do not affect the operational characteristics of the program or, more importantly, the difficulty of development, and should not be counted. Such a count of variables will be referred to as VARS. Thus, for the program payday appearing in the Fig. 6.2 VARS = 7. For the sample program

SORT in Fig. 4.2 VARS = 6 (from a cross-reference listing that identified X, N, I, J, SAVE and IM1 as the variables in the program). The count of variables VARS depends on the following definition:

A variable is a string of alphanumeric characters that is defined by a developer and that is used to represent some value during either compilation or execution.

```
1.   #include < stdio. h >
2.   struct check
3.     {
4.       float gross, tax, net;
5.     } pay;
6.     float hours, rate;
7.     void main ( )
8.       {
9.       while (! feof (stdin))
10.        {
11.          scanf("%f %f", & hours, & rate);
12.          pay. gross = hours * rate;
13.          pay. tax = 0.25 * pay. gross;
14.          pay. net = pay. gross - pay. tax;
15.          printf("%f %f %f/n", pay. gross, pay. tax, pay. net);
16.        }
17.      }
```

Fig. 6.2: Payday program

check	2				
gross	4	12	13	14	15
hours	6	11	12		
net	4	14	15		
pay	5	12	13	13	14
	14	14	15	15	15
rate	6	11	12		
tax	4	13	14	15	

Fig. 6.3: A cross reference of program payday

Although a simple way to obtain VARS is from a cross-reference list, it can also be generated using a software analyzer that counts the individual tokens [CONT86].

While it may sound simple to determine the value of VARS — certainly, counting the number of variables in a program seems straightforward—there are some items in the cross-reference listing of the program in Fig. 6.2 that have been deleted. These items are listed in Fig. 6.4. The items feof stdin are related to I/O. The cross-referencing software called the name of the program payday a variable. However, because none of these are variables in the sense of variables that we create to produce a program, we deleted them from Fig. 6.3, but it should be clear that this "algorithmic" metric VARS is, in fact, a little subjective. In determining this metric, as with all other software metrics including lines of code, we attempt to establish guidelines that eliminate as much subjectivity as possible. But, the reader is well advised to realize that total objectivity is impossible.

feof	9
stdin	10

Fig. 6.4: Some items not counted as VARS

Among all the variable names in line 13 of Fig. 6.2 is the constant 0.25. This program assumes that all pay will be taxed at a 25% rate. Also, in line 05 of Fig. 4.2 of chapter 4, the constant 2 is used to avoid sorting arrays with less than two elements. None of these constants 0.25 or 2 are counted as VARS, and yet they play special purposes in the programs. Further more, mathematical constants such as τ and ε are important for programs involving trigonometric or logarithmic applications. Even array references with an explicit index, such as A[11], may indicate some special meaning for that particular location.

Halstead [HALS77] introduced a metric that he referred to as η_2 to be a count of the operands in a program—including all variables, constants, and labels. Thus,

$$\eta_2 = \text{VARS} + \text{unique constants} + \text{labels}.$$

The sample SORT program in Fig. 4.2 which is analysed in Table 6.2 has 6 variables (X, N, I, J SAVE, IM1), 3 constants (1, 2, 0) and 01 labels (SORT) so that η_2 = 10. The name of the subroutine SORT is treated as a label since it is the label that will be used by any other program or sub-program that wants to access SORT. The program payday in Fig. 6.2 has 7 variables (check, gross, hours, net, pay, rate, tax), 1 constant 0.25, and no label. Thus, its η_2 is 8 note that η_2 is the count of the number of unique operands. Thus, this metric fails to capture an important feature of the "amount of data"—namely, the total operand usage. For example, given the 8 operands in Fig. 6.2, it is possible to construct the program shown or to construct a much larger and more complicated program in order to measure the quantity of usage of the operands. Halstead further defined the metric total occurrence of operands, and named it N_2. Fig. 6.5 repeats Fig. 6.2 and encloses each operand occurrences in brackets [CONT86].

The program payday uses the 8 operands 30 times: some are used several times (like pay) and some are used sparingly (0.25 is used only once). Thus, N_2 = 30 for this program.

The metrics VARS, η_2, and N_2 are the most popular data structure measures. They seem to be robust, slight variations in algorithm computation schemes for computing them do not seem to affect inordinately other measures based upon them.

```
# include < stdio. h >
struct [check]
{
   float [gross], [tax], [net];
} [pay];
float [hours], [rate);
void main ( )
{
while (! feof (stdin))
   {
      scanf("% f % f", & [hour], & [rate]);
      [pay] . [gross] = [hours] * [rate];
      [pay] . [tax] = 0.25 * [pay] . [gross];
      [pay] . [net] = [pay] . [gross] - [pay] . [tax];
      printf("% f % f % f/n", [pay] . [gross] [pay] . [tax], [pay] . [net]);
   }
}
```

Fig. 6.5: Program payday with operands in brackets

6.3.2 The Usage of Data within a Module

In Fig. 6.6 the program "bubble" inputs two related integer arrays (a and b) of the same size up to 100 elements each. It uses a bubble sort on the a-array, interchanges the b-array values to keep them with the accompanying a-array values, and outputs the results. Prior to Fig. 6.6, all of our examples have illustrated small, single-module programs or subroutines. Fig. 6.6 contains a main program in lines 11–37 and a sub-program procedure swap in lines 3–9.

Several metrics may be computed for individual modules. In order to characterize the intra-module data usage, we may use the metrics live variables and variable spans that are discussed below.

Live variables: While constructing program "bubble", the developer created a variable "last". Analyse Fig. 6.6 carefully to see that all array elements beyond the "last" one are sorted. While the program is running, if size = 25 and last = 14, then all items a[15]–a[25] and b[15]–b[25] are in order even though the first 14 elements of each array are not yet sorted. A beginning value for "last" is established in line 17, decremented in line 22, and used in the logical expression in line 24.

There are only three statements in this program in which "last" appears, excluding the declaration in line 13. Does this mean that we do not need to be concerned with "last" while constructing the statements other than 17, 22, and 24? Certainly not. Between statements 17 and 24, it is important to keep in mind what "last" is doing. For example statements 18–19 are

used to set up a potentially never-ending loop. However, even though these statements never mention "last", the developer realized that each time on a-value "bubbles down" to its appropriate position. "Last" will be decremented by one. Eventually on some cycle through the a-values none will be swapped and the loop beginning in statement 19 will be exited. As we will show later, "last" has life span that begins at statement 17 and extends through statement 24.

Thus, a developer must constantly be aware of the status of a number of data items during the development process. A reasonable hypothesis is: more the data items that a developer must keep track of when constructing statements, the more difficult it is to construct. Thus, our interest lies in the size of the set of those data items called live variables (LV) for each statement in the program.

As suggested earlier, the set of live variables for a particular statement is not limited to the number of variables referenced in that statement. For example, the statement being considered may be just one of the several that set up the parameters for a complex procedure. The developer must be aware of entire list of parameters to know that they are being set up in an orderly fashion, so that any statement in the group disturbs no variables later. Therefore, there are several possible definitions of a live variable [DUNS579].

1. A variable is live from the beginning of a procedure to the end of the procedure.

2. A variable is live at a particular statement only if it is referenced a certain number of statements before or after that statement.

3. A variable is live from its first to its last references within a procedure.

The first definition, while computationally simple, does not correspond to the idea of the live variable. According to this definition, both the variable "last" with a 8-statement life span (lines 17–24) and the variable "size" with a 22 statement life span (lines 14–35) can be considered alive throughout the procedure. The second definition might work, but there is no agreement on what a "certain number of statements" should be and no successful use has been reported.

The third definition meets the spirit of the live variable idea and is easy to compute algorithmically. In fact, a computer program (a software analyzer) can produce live variable counts for all statements in a program or procedure.

1	`#include < stdio. h >`
2	
3	`void swap (int x [ ], int K)`
4	`{`
5	`int t;`
6	`t = x[K];`
7	`x[K] = x[K + 1];`
8	`x[K + 1] = t;`
9	`}`

(Contd.)...

```
void main ( )
{
int i, j, last, size, continue, a[100], b[100];
scanf("% d", & size);
for (j = 1; j < = size; j + +)
scanf("%d %d", & a[j], & b[j];
last - size;
continue = 1;
while(continue)
{
   continue = 0;
   last = last-1;
   i = 1;
while (i < = last)
{
   if (a[i] > a[i + 1])
   {
      continue = 1;
      swap (a, i);
      swap (b, i);
   }
      i = i + 1;
}
}
for (j = 1; j < = size; j ++)
printf("%d %d\n", a[j], b[j]);
}
```

Fig. 6.6: Bubble sort program

It is thus possible to define the average number of live variables ($\overline{LV}$), which is the sum of the count of live variables divided by the count of executable statements in a procedure. This is a complexity measure for data usage in a procedure or program. The live variables in the program in Fig. 6.6 appear in Fig. 6.7 the average live variables for this program is $\frac{124}{34} = 3.647$.

Line	*Live variables*	*Count*
4	—	0
5	—	0
6	t,x,k	3
7	t,x,k	3
8	t,x,k	3
9	—	0
10	—	0
11	—	0
12	—	0
13	—	0
14	size	1
15	size, *j*	2
16	size, *j*, *a*, *b*	4
17	size, *j*, *a*, *b*, last	5
18	size, *j*, *a*, *b*, last, continue	6
19	size, *j*, *a*, *b*, last, continue	6
20	size, *j*, *a*, *b*, last, continue	6
21	size, *j*, *a*, *b*, last, continue	6
22	size, *j*, *a*, *b*, last, continue	6
23	size, *j*, *a*, *b*, last, continue, *i*	7
24	size, *j*, *a*, *b*, last, continue, *i*	7
25	size, *j*, *a*, *b*, continue, *i*	6
26	size, *j*, *a*, *b*, continue, *i*	6
27	size, *j*, *a*, *b*, continue, *i*	6
28	size, *j*, *a*, *b*, continue, *i*	6
29	size, *j*, *a*, *b*, *i*	5
30	size, *j*, *a*, *b*, *i*	5

(Contd.)...

31	size, *j*, *a*, *b*, *i*	5
32	size, *j*, *a*, *b*, *i*	5
33	size, *j*, *a*, *b*	4
34	size, *j*, *a*, *b*	4
35	size, *j*, *a*, *b*	4
36	*j*, *a*, *b*	3
37	—	0

Fig. 6.7: Live variables for the program in Fig. 6.6

As shown live variables depend on the order of statements in the source program, rather than the dynamic execution-time order in which they are encountered. A metric based on run-time order would be more precisely related to the life of the variable, but would be much more difficult to define algorithmically (especially in a non-structured programming language).

Variable spans: Two variables can be alive for the same number of statements, but their use in a program can be markedly different. For example, Fig. 6.8 lists all of the statements in a C program that refer to the variables "a" and "b". Both variables are alive for the same 40 statements (21–60), but "a" is referred to three times while "b" is mentioned only once. A metric that captures some of the essence of how often a variable is used in a program is called the span (SP). This metric is the number of statements between two successive references of the same variable [ELSH76]. The span is related to the third definition of live variables. For a program that references a variable in n statements, there are n-1 spans for that variable. Thus, in Fig. 6.8 "a" has 4 spans and "b" has only 2. Intuitively this tells us that 'a' is being used more than 'b'.

```
...
21      scanf (" %d %d," & a, & b);
...
32      x = a;
...
45      y = a – b;
...
53      z = a;
...
60      printf (" %d %d," a, b);
...
```

Fig. 6.8: Statements in ac program referring to variables a and b

Furthermore, the size of a span indicates the number of statements that pass between successive uses of a variable. A large span can require the developer to remember during the construction process a variable that was last used in the program. In Fig. 6.8 "*a*" has 4 spans of 10, 12, 7, and 6, statements, while for "*b*" has 2 spans of 23 and 14 statements. It is simple to extend this metric to "average span size", ($\overline{\text{SP}}$) in which case "*a*" has an average span size of 8.75 and '*b*' has an average span size of 18.5.

Making program-wide metrics from intra-module metrics: Each of the metric discussed in this section is intended to be used within a module, as indicated. But it is possible to extend each one into an inter-module metric. For example if we want to characterize the average number of live variables for a program having modules, we can use this equation.

$$\overline{LV}_{program} = \frac{\sum_{i=1}^{m} \overline{LV}_i}{m}$$

where $\overline{LV}_i$ is the average live variable metric computed from the ith module.

Furthermore, the average span size $(\overline{SP})$ for a program of n spans could be computed by using the equation.

$$\overline{SP}_{program} = \frac{\sum_{i=1}^{n} \overline{SP}_i}{n}$$

6.3.3 Program Weakness

A program consists of modules. Using the average number of live variables $(\overline{LV})$ and average life of variables (γ), the module weakness has been defined as [YOGE98]:

$$WM = \overline{LV} * \gamma$$

Average number of live variables and average life of variables can be found using some automated tools. Even most compilers and assemblers have an option to generate a cross-reference list, indicating the line number where a certain variable is declared and the line or lines where it is referenced. Such a list may be used to compute the value of LV and γ. Using these two values, weakness of a module can be computed. The weakness of the module can be used to estimate the testability and maintainability. If weakness of a module is more, testability will be better and vice-versa. Weakness will also have effect on maintainability. A weaker module will be more difficult to maintain.

As we all know a program is normally a combination of various modules, hence program weakness can be a useful measure and is defined as:

$$WP = \frac{\left(\sum_{i=1}^{m} WM_i\right)}{m}$$

where, WM_i : weakness of ith module.

WP : weakness of the program

m : number of modules in the program.

Example 6.3

Consider a program for sorting and searching. The program sorts an array using selection sort and than search for an element in the sorted array. The program is given in Fig. 6.8. Generate cross–reference list for the program and also calculate $\overline{LV}$, γ, and WM for the program.

Solution

The given program is of 66 lines and has 11 variables. The variables are *a, i, j,* item, min, temp, low, high, mid, loc and option.

```
/**************************************************************************/
/**** PROGRAM TO SORT AN ARRAY USING SELECTION SORT & THEN SEARCH
*******/
/**** FOR AN ELEMENT IN THE SORTED ARRAY ********/
/**************************************************************************/

#include <stdio.h>
#define MAX 10

main ()
{
  int a[MAX];
  int i,j,item,min,temp;
  int low=0,high,mid,loc;
  char option;

  for (i=0;i<MAX;i++)
    {
    printf("Enter a[%d]:",i);
    scanf("%d",&a[i]);
    }
  /* selection sort */
    for (i=0;i<(MAX-1);i++)
    {
    min=i;
    for (j=i+1; j<MAX;j++)
      {
      if (a[min]>a[j])
        {
        temp=a[min];
        a[min]=a[j];
        a[j]=temp;
```

(Contd.)...

```
printf("\n The Sorted Array:\n");
for (i=0; i<MAX;i++)
  printf("\n a[%d]=%d",i,a[i]);
printf("\n Do you want to search any element in the array
                                              (Y/N):");

fflush(stdin);
scanf("%c",& option);
if (toupper(option)=='Y')
   {
   printf("\n Enter the item to be searched :");
   scanf("%d", &item);
   high=MAX;
   mid=(int)(low+high)/2;
   while ((low<=high)&&(item!=a[mid]))
   {
     if (item>a[mid])
     low=mid+1;
     else high=mid-1;
     mid=(int) (low+high)/2;
   }
   if(low>high)
   {
     loc=0;
     printf("\n No such item is present in the array\n");
   }
   if (item==a[mid])
   {
     loc=mid;
     printf("\n The item %d is present at location %d in the sorted
     array\n",item,loc);
   }
  }
 printf("\n Sorting & Searching done");
}
```

Fig. 6.8: Sorting and searching program

Cross-Reference list of the program is given below:

a	11	18	19	27	27	29	30	30	31	37	47	49	59		
i	12	16	16	16	18	19	22	22	22	24	36	36	36	37	37
j	12	25	25	25	27	30	31								
item	12	44	47	49	59	62									
min	12	24	27	29	30										
temp	12	29	31												
low	13	46	47	50	52	54									
high	13	45	46	47	51	52	54								
mid	13	46	47	49	50	51	52	59	61						
loc	13	56	61	62											
option	14	40	41												

Live variables per line are calculated as:

Line number	*Live variables on the line*	*Count*
13	low	1
14	low	1
15	low	1
16	low, *i*	2
17	low, *i*	2
18	low, *i*, *a*	3
19	low, *i*, *a*	3
20	low, *i*, *a*	3
22	low, *i*, *a*	3
23	low, *i*, *a*	3
24	low, *i*, *a*, min	4
25	low, *i*, *a*, min, *j*	5
26	low, *i*, *a*, min, *j*	5
27	low, *i*,*a*, min, *j*	5
28	low, *i*, *a*, min, *j*	5
29	low, *i*, *a*, min, *j*, temp	6
30	low, *i*, *a*, min, *j*, temp	6
31	low, *i*, *a*, *j*, temp	5
32	low, *i*, *a*,	3
33	low, *i*, *a*	3
34	low, *i*, *a*	3
35	low, *i*, *a*	3

(Contd.)...

36	low, i, a	3
37	low, i, a	3
38	low, a	2
39	low, a	2
40	low, a, option	3
41	low, a, option	3
42	low, a	2
43	low, a	2
44	low, a, item	3
45	low, a, item, high	4
46	low, a, item, high, mid	5
47	low, a, item, high, mid	5
48	low, a, item, high, mid	5
49	low, a, item, high, mid	5
50	low, a, item, high, mid	5
51	low, a, item, high, mid	5
52	low, a, item, high, mid	5
53	low, a, item, high, mid	5
54	low, a, item, high, mid	5
55	a, item, mid	3
56	a, item, mid, loc	4
57	a, item, mid, loc	4
58	a, item, mid, loc	4
59	a, item, mid loc	4
60	item, mid, loc	3
61	item, mid, loc	3
62	item, loc	2
63		0
64		0
65		0
66		0
	Total	174

Thus Avg. number of Live Variables $(\overline{\text{LV}}) = \dfrac{\text{Sum of count of live variables}}{\text{Count of executable statements}}$

$$\overline{\text{LV}} = \frac{174}{53} = 3.28$$

$$\Rightarrow \qquad \overline{\text{LV}} = 3.28$$

$$\gamma = \frac{\text{Sum of count of live variables}}{\text{Total no. of variables}}$$

$$\Rightarrow \quad \gamma = \frac{174}{11} = 15.8$$

$$\Rightarrow \quad \gamma \ (i.e. \text{ Avg. life of variables}) = 15.8$$

Module weakness

$$WM = \overline{LV} \times \gamma$$

where WM is the module weakness

$\overline{LV}$ is the Avg. no. of live variables

& γ is the Avg. life of variables

$$\Rightarrow \quad WM = 3.28 \times 15.8 = 51.8$$

$$WM = 51.8$$

Example 6.4

Consider a program given in Fig. 6.9 that draws a circle using midpoint algorithm. Generate cross reference list of variables and also calculate average number of live variables ($\overline{LV}$), average life of variables (γ) and program weakness (WM).

Solution

There are 9 variables declared in the program. The variables are rad, p_0, p_1, x, y, x_c, y_c, d and m.

```
\\To scan convert a circle using midPoint Algorithm
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
void circle (int, int, int, int);
void  main ()
{
int rad;
int p0, p1;
int x, y;
int xc, yc;
int d,m;
d = DETECT;
initgraph (&d, &m," ");
setbkcolour(BLACK);
clrscr ();
cout <<" enter the radius of circle:";
cin >> rad;
cout <<"Enter the value of center co-ordinates:";
```

(Contd.)...

```
cin >> xc >> yc;
x = 0;
y = rad;
circle(xc, yc, x, y);
p0 = 1 - rad;
while (x <y)
{
if(p0 <0)
{
x ++ ;
p0 = p0 + 2*(x + 1) + 1;
circle (x, y, xc, yc);
}
else
{
x ++;
y --;
p0 = p0 + 2* (x - y) + 1;
circle(x, y, xc, yc);
}
}
getch();
}
void circle(int x, int y, int xc, int yc)
{
putpixel(xc + x, yc + y, 4);
putpixel(xc - x, yc + y, 4);
putpixel(xc + x, yc - y, 4);
putpixel(xc - x, yc - y, 4);
putpixel(xc + y, yc + x, 4);
putpixel(xc - y, yc + x, 4);
putpixel(xc + y, yc - x, 4);
putpixel(xc - y, yc - x, 4);
}
```

Fig. 6.9: Program for drawing a circle using midpoint algorithm

The cross reference list is given below:

Variable	*Reference a Line Number*
rad	7, 17, 21, 23,
p_0	8, 23, 26, 29, 29,36, 36,
p_1	8
x	9, 20, 22, 24, 28, 29, 30, 34, 36, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
y	9, 21, 22, 24, 30, 35, 36, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
x_c	10, 19, 22, 30, 37, 42, 44, 45, 46. 47, 48, 49, 50, 51
y_c	10, 19, 22, 30, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
d	11, 12, 13
m	11, 13

Live variables per line are given below:

Line number	*Live variables*	*Count*
12	d	1
13	d,m	2
14	–	0
15	–	0
16	–	0
17	rad	1
18	rad	1
19	rad, x_c, y_c	3
20	rad, x_c, y_c,x	4
21	rad, x_c, y_c, x, y	5
22	rad, x_c, y_c, x, y	5
23	rad, x_c, y_c, x, y, p_0	6
24	x_c, y_c, x, y, p_0	5
25	x_c, y_c, x, y, p_0	5
26	x_c, y_c, x, y, p_0	5
27	x_c, y_c, x, y, p_0	5
28	x_c, y_c, x, y, p_0	5
29	x_c, y_c, x, y, p_0	5
30	x_c, y_c, x, y, p_0	5
31	x_c, y_c, x, y, p_0	5
32	x_c, y_c, x, y, p_0	5
33	x_c, y_c, x, y, p_0	5
34	x_c, y_c, x, y, p_0	5
35	x_c, y_c, x, y, p_0	5

(Contd.)...

36	x_c, y_c, x, y, p_0	5
37	x_c, y_c, x, y	4
38	x_c, y_c, x, y	4
39	x_c, y_c, x, y	4
40	x_c, y_c, x, y	4
41	x_c, y_c, x, y	4
42	x_c, y_c, x, y	4
43	x_c, y_c, x, y	4
44	x_c, y_c, x, y	4
45	x_c, y_c, x, y	4
46	x_c, y_c, x, y	4
47	x_c, y_c, x, y	4
48	x_c, y_c, x, y	4
49	x_c, y_c, x, y	4
50	x_c, y_c, x, y	4
51	x_c, y_c, x, y	4
52		0
	Total	153

$$\overline{LV} = \text{Average number of live variables}$$

$$= \frac{\text{Sum of count of live variables}}{\text{Count of executable statements}}$$

$$= \frac{153}{41} = 3.73$$

$$\gamma = \text{Average life of variables}$$

$$= \frac{\text{Sum of count of live variables}}{\text{Number of unique variables}}$$

$$= \frac{153}{9} = 17$$

Program weakness $= \overline{LV} \times \gamma$

$$= 3.73 \times 17 = 63.41.$$

6.3.4 The Sharing of Data Among Modules

As discussed earlier, a program normally contains several modules and share coupling among modules. However, it may be desirable to know the amount of data being shared among the modules.

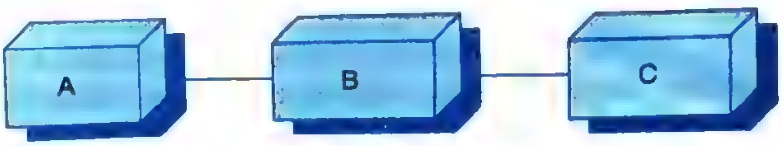


Fig. 6.10: Three modules from an imaginary program

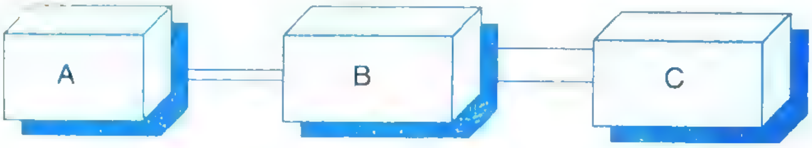


Fig. 6.11: "Pipes" of data shared among the modules

For example, Fig. 6.10 shows a schematic of a program consisting of the three subprograms A, B, and C. If we include information that represents the amount of data "passed" among the subprograms, we could envision pipes between the subprograms. The diameter of each pipe could represent the quantity or volume, of data sent from one subprogram to be used in the other. Fig. 6.11 shows the same three subprograms with pipes representing data shared between A and B and between B and C. Note that the A-B shared data is implicitly less than the B-C shared data. Fig. 6.12 shows a pictorial representation of the data shared between the main program of bubble and procedure swap both from Fig. 6.6 in bubble, the main program invokes the procedure swap in order to get it to swap the ith and $(i + 1)$ th members of the a-vector and the b-vector.

Here, we will introduce metrics that can be used for measuring this concept of sharing of data between modules. Keep in mind that the "bigger the pipe" in between any two modules, the more complex is their relationship. In theory, every module in a program is related to every other module.

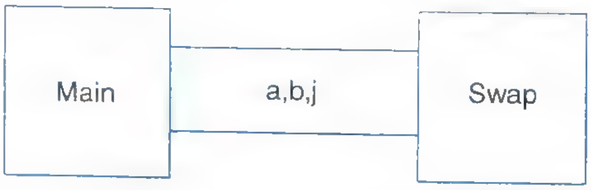


Fig. 6.12: The data shared in program bubble

If this were not true, the modules would not have been put into the same program. But, in practice, some modules may share no data directly with another, as shown by modules A and C in Fig. 6.11 The data structure employed in A should have little effect on module C, while the data structure employed in B is obviously important to C since they have such a large pipe between them.

The relationship between modules is simple if two variables are passed between them. However, a module with long list of parameters of different types, and some "global" data structure, which is shared, with several modules should be more difficult to construct or comprehend.

We assume that a global variable is one that is available to any and all modules in a program. Most programming languages allow the declaration of variables that can be accessed anywhere in the program. Contrast a global variable with a local variable, which is declared in a specific module, and whose name is unknown outside that module. A local variable is available to another module only when specifically mentioned in a parameter list passed to that module (in a procedural language), or when referred to in a module completely contained in the one where the local variable is declared (in a block-structured language). Global variables are not so limited, and are known and usable everywhere in the program.

6.4 INFORMATION FLOW METRICS

The other set of metrics we would like to consider are generally known as 'Information Flow' (IF) metrics. The basis of IF metrics is found upon the following premise. All but the simplest systems consist of components, and it is the work that these components do and how they are fitted together that influences the complexity of a system. If a component has to do numerous discrete tasks, it is said to lack 'cohesion'. If it passes information to, and/or accepts information from many other components within the system, it is said to be highly 'coupled'. Systems theory tells us that components that are highly coupled and that lack cohesion tend to be less reliable and less maintainable than those that are loosely coupled and that are cohesive. The following are the working definitions of the terms used above:

Component	:	Any element identified by decomposing a (software) system into its constituent parts.
Cohesion	:	The degree to which a component performs a single function.
Coupling	:	The term used to describe the degree of linkage between one component to others in the same system.

This systems-view map to software systems is extremely easy to understand as most engineers today use, or are at least familiar with, top down design techniques that produce a hierarchical view of system components. Even the more modern 'middle out' design approaches produce this structured type of deliverable, and here again IF metrics can be used.

Information flow metrics model the degree of cohesion and coupling for a particular system component. How that model is constructed can justifiably range from the simple to the complex. We intend to start with the most simple representation of IF metrics to illustrate the basic concepts, how to derive information using the metrics and how to use that information.

In terms of applying IF metrics to software systems, the pioneering work was done by Henry and Kafura [HENR81]. They looked at the UNIX operating system, and found a strong association between the IF metrics and the level of maintainability ascribed to components by developers. Other individuals who tried to apply these principles found difficulties in using the Henry and Kafura approach. Further work was done in the UK by Professor Darrell Ince and Martin Shepperd [INCE89], among others, which resulted in a more practical IF model. This work was complimented by Barbara Kitchenham [KITC90], who addressed the same problem, and who also presented a clear approach to the question of interpretation.

6.4.1 The Basic Information Flow Model

Information flow metrics are applied to the components of a system design. Fig. 6.13 shows a fragment of such a design, and for component 'A' we can define three measures, but remember that these are the simplest models of IF.

1. 'FAN IN' is simply a count of the number of other components that can call, or pass control, to Component A.

2. 'FANOUT' is the number of components that are called by component A.

3. This is derived from the first two by using the following formula. We will call this measure the INFORMATION FLOW index of component A, abbreviated as IF(A).

$$IF(A) = [FAN\ IN(A) \times FAN\ OUT\ (A)]^2.$$

The formula includes a power component to 'model the non-linear nature of complexity' as most texts of IF metrics describe it. (The assumption is that if something is more complex than something else, then resultant is much more complex). Given that assumption, we could raise to a power three or four or whatever we want but, on the principle that the simpler the model the better, then two is a good enough choice. In our view, raising to two makes it easier, as will be seen, to pick out the potential bad guys [GOOD93].

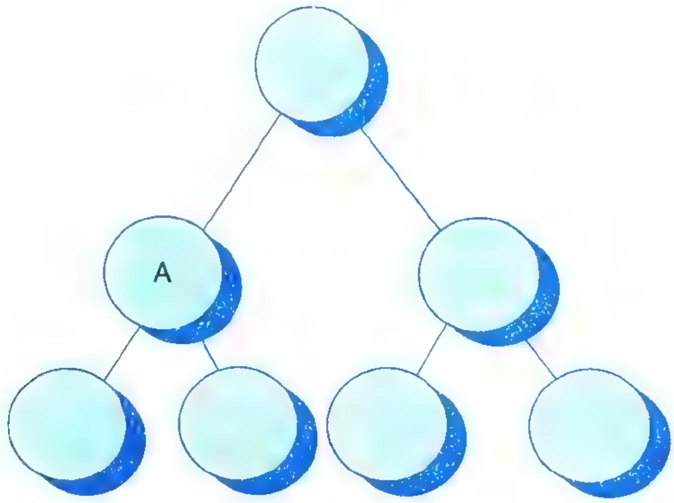


Fig. 6.13: Aspects of complexity

Given functional decomposition it will be seen that there is one additional attribute possessed by each component, namely its level in the decomposition. The following is a step-by-step guide to deriving these most simple of IF metrics.

1. Note the level of each component in the system design.
2. For each component, count the number of calls to that component — this is the FAN IN of that component. Some organizations allow more than one component at the highest level in the design, so for components at the highest level which should have a FAN IN of zero, assign a FAN IN of one. Also note that a simple model of FAN IN can penalize reused components.
3. For each component, count the number of calls from that component. For components that call no other, assign a FAN OUT value of one.
4. Calculate the IF value for each component using the above formula.
5. Sum the IF value for all components within each level which is called as the LEVEL SUM.
6. Sum the IF values for the total system design which is called the SYSTEM SUM.
7. For each level, rank the components in that level according to FAN IN, FAN OUT and IF values. Three histograms or line plots should be prepared for each level.
8. Plot the LEVEL SUM values for each level using a histogram or line plot.

This may sound like a great deal of work, but for most commercial systems, provided the documentation exits, this data can be derived and the analysis done within one engineering day. If the systems are larger, then it will obviously take longer, but remember that once done, it is very easy to keep up-to-date. Depending upon the environment it may even be possible to automate the calculations.

Having got the information, we now need to utilize it. It must be realized that, for IF metrics, there are no absolute values of good or bad. Information Flow metrics are relative indicators. This means that the value for one system may be higher than the other system, but this does not mean that one system is worse. Nor does a high metric value guarantee that a component will be unreliable and un-maintainable. It is only that it will probably be less reliable and maintainable than its fellows.

The rub is that, in most systems, less reliable and less maintainable means that it is potentially going to cost significant amounts of money to fix and enhance. Potentially, it could even be a nightmare component.

A nightmare component is the one that the system administrator has nightmares about, because he or she knows that if anyone touches that component, the whole system is going to crash, and it will take weeks to fix because designers have already left and no one is available to guide.

So the strength of IF metrics is not in the numbers themselves, but in how the information is used. As a guide, 25 per cent of components with the highest scores for FAN IN, FAN OUT and IF values should be investigated. Now in practice it may well be that a certain number of components stick out like a sore thumb, especially on the IF values. If this group is more or less than the 25 per cent guide, then do not worry about it, concentrate on those that seem to be odd according to the metric values rather than following any 25 per cent rule slavishly.

High FAN IN values indicate components that lack cohesion. It may well be that the functions have not been broken out to a great enough degree. Basically, these components are often called because they are doing more than one job.

High levels of FAN OUT also indicate a lack of cohesion or missed levels of abstraction. Here design was stopped before it was finished, and this is reflected in the high number of calls from the component. Generally speaking, FAN OUT appears to be a better indicator of problem components than FAN IN, but it is early days yet and we would not wish to discount FAN IN.

High IF values indicate highly coupled components. These components need to be looked at in terms of FAN IN and FAN OUT to see how to reduce the complexity level. Sometimes a 'traffic centre' may be hit. This is a component where, for whatever reasons, there is a high IF value, but things cannot be improved. Switching components often exhibit this. Here there is a potential problem area which, it is also a large component, may be very error prone. If the complexity cannot be reduced, then at least make sure that component is thoroughly tested.

Looking at the LEVEL SUM plot of values, we should see a fairly smooth curve showing controlled growth in IF across the levels. Sudden increases in these values across levels can indicate a missed level of abstraction within the general design. For systems where the design has less than ten levels, then a simple count of components at each level seems to work equally well.

The final item of information flow metrics is the SYSTEM SUM value. This gives an overall complexity rating for the design in terms of IF metrics. Most presentations on this topic will say that this number can be used to assess alternative design proposals.

6.4.2 A More Sophisticated Information Flow Model

We have looked at the most simple form of IF metrics, but the original proposals put forward by Henry and Kafura [HENR81] were more sophisticated than the control flow-based variant discussed above. As mentioned earlier, Ince and Shepperd [INCE89] and Kitchenham [KITC90] have done a great deal of work to help in the practical application of Henry and Kafura's pioneering proposals, and it is a distillation of that work, that has been summarized by Goodman [GOOD93] into the more sophisticated IF model. It should, however, be realized that this is a model, and it will need to be tailored to one's organization's design mechanisms before it is used. Such a tailoring process should not take more than two days for counting rule derivation and documentation of these rules, provided a well-defined design notation is used together with a competent engineer who knows that notation.

The only difference between the simple and the sophisticated IF models lies in the definition of FAN IN and FAN OUT.

For a component A let:

a = the number of components that call A.

b = the number of parameters passed to A from components higher in the hierarchy;

c = the number of parameters passed to A from components lower in the hierarchy;

d = the number of data elements read by component A.

Then:

$$\text{FAN IN(A)} = a + b + c + d$$

Also let:

e = the number of components called by A;

f = the number of parameters passed from A to components higher in the hierarchy;

g = the number of parameters passed from A to components lower in the hierarchy;

h = the number of data elements written to by A.

Then:

$$\text{FAN OUT (A)} = e + f + g + h$$

Other than those changes to the basic definitions, the derivation, analysis and interpretation remain the same. It is advisable for any organization starting to apply IF metrics to build up confidence by using the simpler form. If these work, then leave it at that. If, and only if, the simpler form fails in environment, in other words we feel confident that no significant relationship exists between the simple measures and the levels of reliability and maintainability, then spend the effort to tailor and pilot the more sophisticated form.

It is encouraging to know that there have been a number of experimental validations of IF metrics that seem to support the claims made for them. Programming groups that have been introduced to IF metrics have been able to make use of them and they also report benefits in the area of design, quality control and system management. They seem to work, but there appears to be some reluctance in the industry as a whole to make use of IF metrics. Perhaps one reason is that managers feel they are a bit 'techie'. Perhaps others feel that they are not yet ready to use sophisticated techniques like IF metrics.

6.5 OBJECT ORIENTED METRICS

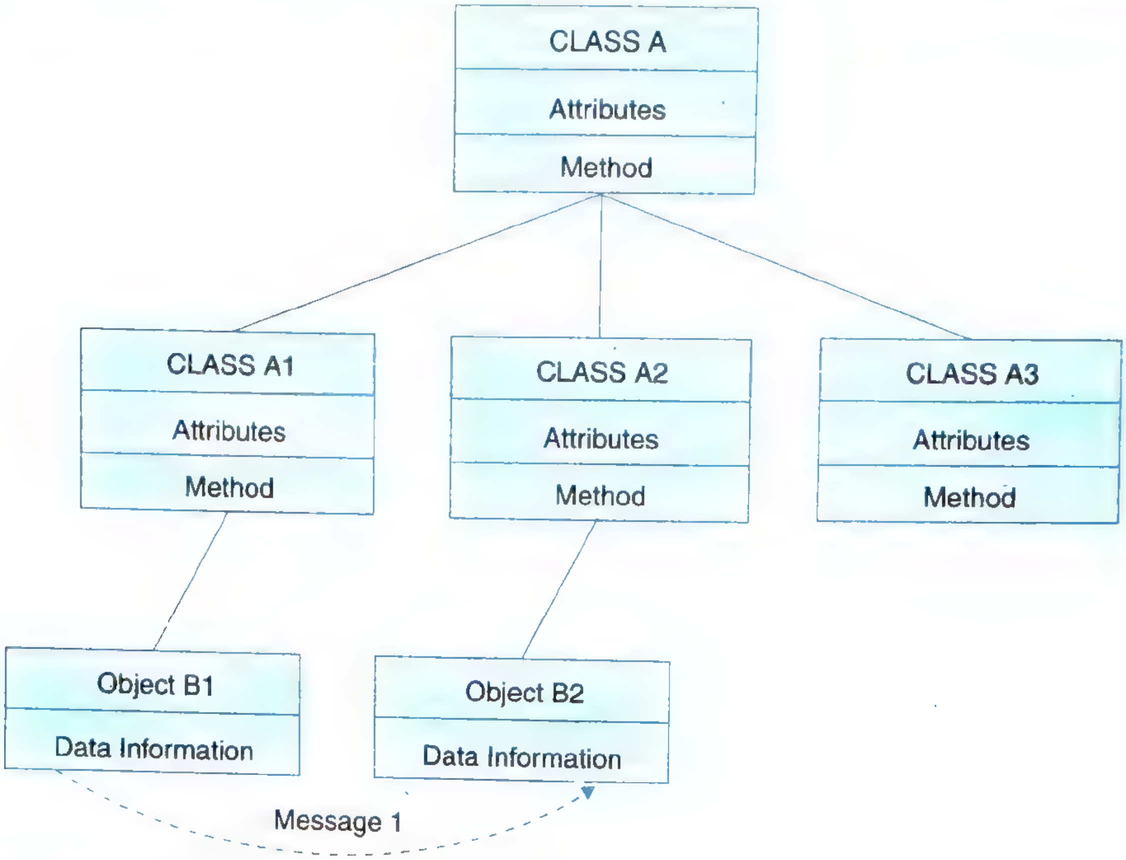
Object oriented design and development has become a popular way of software development. Traditional metrics may not be directly applicable in this area. Hence, a new set of metrics has been developed to fulfil the needs of developers, practioners, researchers and quality controllers. Some of the terminologies are given in Table 6.5, and are very common in object oriented metrics. The pictorial view of few terms are given in Fig. 6.14 [ROSE 97]. The metrics are divided into several categories viz. size, coupling, cohesion, inheritance etc. [ARV106, SHYA91, SHYA94].

6.5.1 Size Metrics

The size metrics given here measure the size of the system in terms of attributes and methods included in the class and capture the complexity of the class.

Table 6.5 Some terminologies used in object oriented metrics

S. No.	*Term*	*Meaning / Purpose*
1.	Object	Object is an entity able to save a state (information) and offers a number of operations (behaviour) to either examine or affect this state.
2.	Message	A request that an object makes of another object to perform an operation.
3.	Class	A set of objects that share a common structure and common behaviour manifested by a set of methods; the set serves as a template from which object can be created.
4.	Method	An operation upon an object, defined as part of the declaration of a class.
5.	Attribute	Defines the structural properties of a class and unique within a class.
6.	Operation	An action performed by or on an object, available to all instances of class, need not be unique.
7.	Instantiation	The process of creating an instance of the object and binding or adding the specific data.
8.	Inheritance	A relationship among classes, wherein an object in a class acquires characteristics from one or more other classes.
9.	Cohesion	The degree to which the methods within a class are related to one another.
10.	Coupling	Object A is coupled to object B, if and only if A sends a message to B.



Note: 1 class A1 is a child class of A and inherits attributes and methods of A.
2. Object B1 contains data and structure from class A1 and class A through inheritance.
3. Object B2 contains data and structure from class A2 and class A through inheritance.
4. Object B1 passes a message 1 to object B2. Hence class A1 is coupled to class A2 through message 1.

Fig. 6.14: Pictorial view of few object oriented terms

(*a*) **Number of attributes per class (NOA):** It counts the number of attributes defined in a class. Fig. 6.15 shows the class diagram of book information system. In this system, Number of Attributes (NOA) for publication class is 2. So NOA = 2 for publication class.

(*b*) **Number of methods per class (NOM):** It counts the number of methods defined in the class. In Fig. 6.15, class publication has two methods getdata ( ) and display ( ). Hence NOM = 2 for publication class.

(*c*) **Weighted methods per class (WMC):** The WMC is the count of sum of complexities of all methods in a class. The method complexity in measured using cyclomatic complexity. This should be normalised so that nominal complexity for a method is taken as unity. Consider a class K_1, with methods $M_1, M_2 \ldots\ldots M_n$ that are given in the class. Let $C_1, C_2 \ldots\ldots, C_n$ be the complexities of the methods. WMC is defined as :

$$\text{WMC} = \sum_{i=1}^{n} C_i$$

If method's complexities are nominal (value = 1), then WMC = n, which is equal to number of methods.

In Fig. 6.15, WMC for book is 3, sale is 2 and publication is 2 (considering each method complexity to be unity).

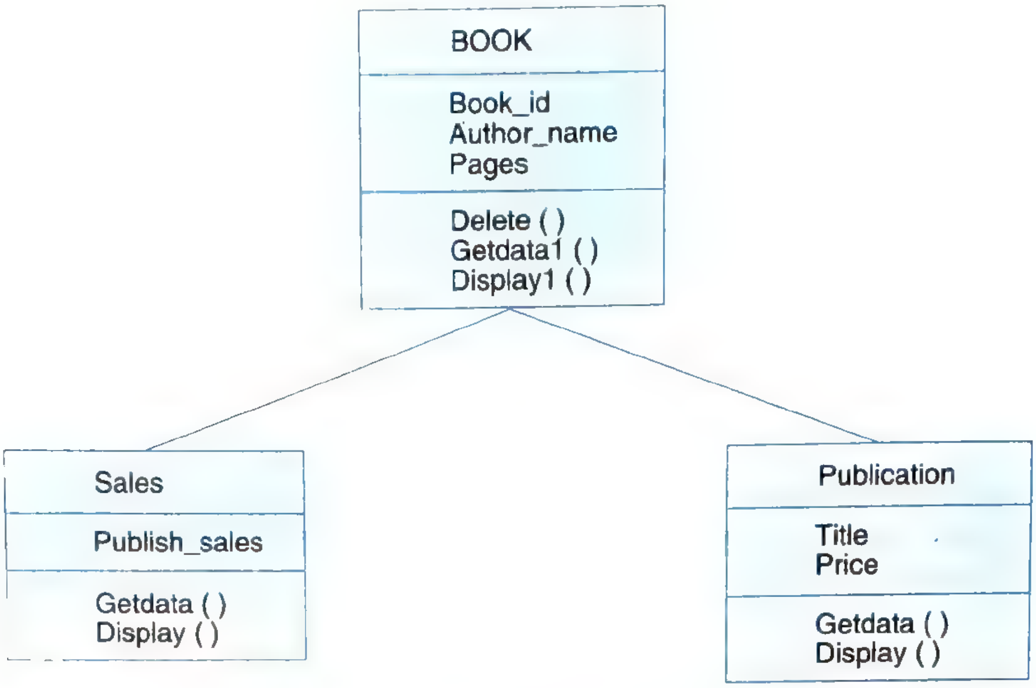


Fig. 6.15: Class diagram of book Information system

(*d*) **Response for a class (RFC):** The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some methods in the class. This includes all methods accessible within class hierarchy. This metric gives us the idea about complexity of a class through number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated. It requires more understanding and effort on the part of testing team. The RFC for class Book (refer Fig. 6.15) is the number of methods that can be invoked in response to messages by itself, by sales class and by publication class. Hence RFC for Book = 3 (self) + 2 (sales) + 2 (publications) = 7.

6.5.2 Coupling Metrics

Coupling relations increase complexity, reduce encapsulation, potential reuse, and limit understanding and maintainability. An improvement of modularity is achieved when inter object class couples in minimized. Evidently, the large number of couples, the higher the sensitivity to changes in other parts of design and less is the possibility of reuse of the class. Some metrics are discussed below :

(*a*) **Coupling between objects (CBO):** CBO for a class is the count of the number of other classes to which it is coupled. Two classes are coupled when methods declared in one class use methods or instance variables by the other class. The excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse in another application. In Fig. 6.16, the Book class contains instances of the classes

publication and sales. The book class delegates its publication and sales issues to instances of the publication and sales classes. The value of metric CBO for class book is 2 and for class publication and class sales is zero.

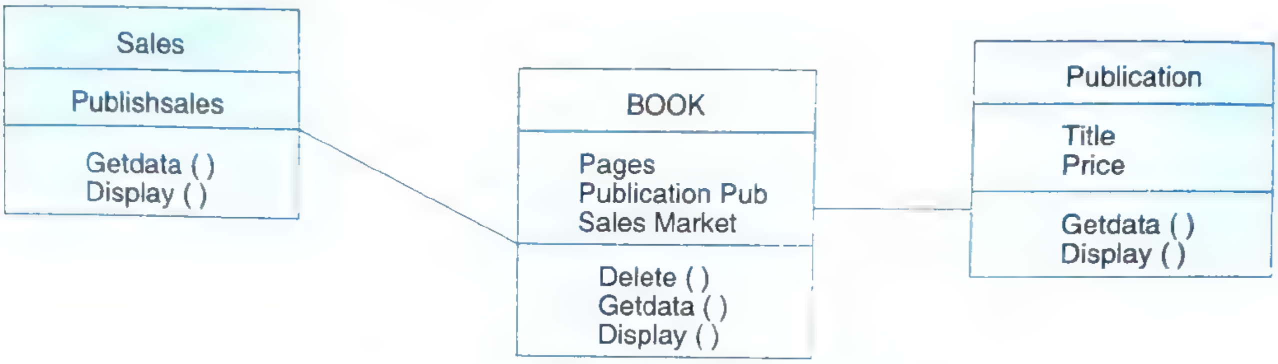


Fig. 6.16: Class diagram of sales information system

(*b*) **Data abstraction coupling (DAC):** It is a technique of creating new data types suited for an application to be developed. It provides the ability to create user defined data types called Abstract Data Types (ADTs) Li and Henry [LI93] defined Data Abstraction Coupling (DAC) as :

DAC = number of ADTs defined in a class.

In Fig. 6.16, there are two ADTs in class book, pub and market. DAC for book class is 2.

(*c*) **Message passing coupling (MPC):** Li and Henry [LI 93] defined Message Passing Coupling (MPC) metric as "number of send statements defined in a class". So, if two different methods in class A access the same method in class B, then MPC = 2. In Fig. 6.16, MPC value of class book is 4 as methods in class book calls sales :: getdata ( ), sales : : display ( ), publication : : getdata ( ), publications : : display ( ).

(*d*) **Coupling factor (CF):** Coupling can be due to message passing (dynamic coupling) or due to semantic association links (static coupling) among class instances.

It is desirable to reduce communication amongst classes and even if they communicate, very little information should be exchanged. It is defined as :

$$CF = \frac{\sum_{i=1}^{TC} \sum_{J=1}^{TC} [\text{Is\_client}(C_i, C_j)]}{TC^2 - TC}$$

where TC is the total number of classes.

$$\text{Is\_client}(C_i, C_j) = \begin{cases} 1 & \text{if } C_i \text{ and } C_j \text{ are coupled} \\ 0 & \text{otherwise} \end{cases}$$

Coupling due to the use of the inheritance is not included in CF, because a class is heavily coupled to its ancestors via inheritance. If no classrd are coupled, CF = 0%. If all classes are coupled with all other classes, CF = 100%.

6.5.3 Cohesion Metrics

Cohesion is a measure of the degree to which the elements of a module are functionally related. A strongly cohesive module implements functionality that is related to one feature of the software and requires little or no interaction with other modules. Thus, we want to maximize the interactions within a module. Four metrics are given here.

(*a*) **Lack of cohesion in methods (LCOM):** Lack of cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes. A highly cohesive module should stand alone ; high cohesion indicates good class subdivision. Classes with low cohesion should probably be subdivided into two or more subclasses with increased cohesion.

Consider a class C_1 with n methods $M_1, M_2 \ldots\ldots M_n$.

Let $\{I_i\}$ = set of all instance variables used by method M_i.

There are n such sets $\{I_1\}, \ldots\ldots \{I_n\}$.

Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \phi\}$ and

$Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \phi\}$

If all n sets $\{I_1\}, \ldots, \{I_n\}$ are ϕ then $P = \phi$.

$$\text{LCOM} = |P| - |Q|, \text{ if } |P| > |Q|$$
$$= 0 \text{ otherwise}$$

In Fig. 6.17, there are four methods M_1, M_2, M_3 and M_4 in class book.

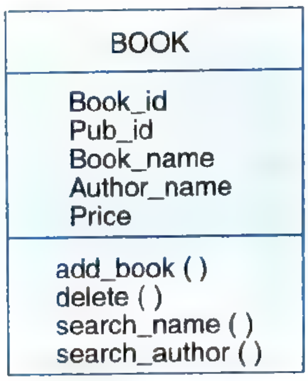


Fig. 6.17: Class diagram of class book in library management system

I_1 {add_book ( )} = {Book_id, Pub_id, Book_name, Author_name, Price}

I_2 {delete ( )} = { Book_id}

I_3 {search_name ( )} = {Book_name}

I_4 {search_author ( )} = {Author_name}

$I_1 \cap I_2$, $I_1 \cap I_3$, $I_1 \cap I_4$ are non-null sets

But $I_2 \cap I_3$, $I_2 \cap I_4$ and $I_3 \cap I_4$ are null sets

LCOM = 0, if number of null interactions are not greater than number of non-null interactions.

Hence, in this case, LCOM = 0 [$|P| = |Q| = 3$]. A positive high value of LCOM implies that classes are less cohesive. Hence, low value of LCOM is desirable.

(*b*) **Tight class cohesion (TCC):** It is defined as the percentage of pairs of public methods of the class with common attribute usage. In Fig. 6.17, methods defined access the following attributes :

add_book ( ) = {Book_id, Pub_id, Book_name, Author_name, Price}

delete ( ) = {Book_id}

Search_name ( ) = {Book_name}

Search_author ( ) = {Author_name}

All methods in class book are public. Number of pairs of methods = 6.

Methods pairs with common attribute usage

= {add_book ( ), delete ( )}, {add_book, search_name}
and {delete ( ), search_author}

Hence, $\text{TCC} = \frac{3}{6} \times 100 = 50.$

(*c*) **Loose class cohesion (LCC):** In addition to direct attributes, this measure considers attributes indirectly used by a method. Method m directly or indirectly invokes a method m', which uses attribute a. LCC is same as TCC except that this metric also considers indirectly connected methods. The LCC is defined as the percentage of pairs of public methods of the class, which are directly or indirectly connected. In Fig. 6.17, LCC for class book is same as TCC *i.e.*, 50% as there are no indirect invocations by the methods of class book.

(*d*) **Information flow based cohesion (ICH):** ICH for a class is defined as the number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method. In Fig. 6.17, method delete ( ) calls function search_name, which in also the method of the same class. Hence value for ICH is 1.

6.5.4 Inheritance Metrics

These metrics are based on the inheritance property of object oriented software. These metrics are easy to calculate and some are given below :

(*a*) **Depth of inheritance tree (DIT):** The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree. In Fig. 6.18, DIT for result class is 2 as it has 2 ancestor classes Internal_Exam/External_Exam and student. DIT for Internal_Exam and External_Exam is 1 as it has one ancestor class student.

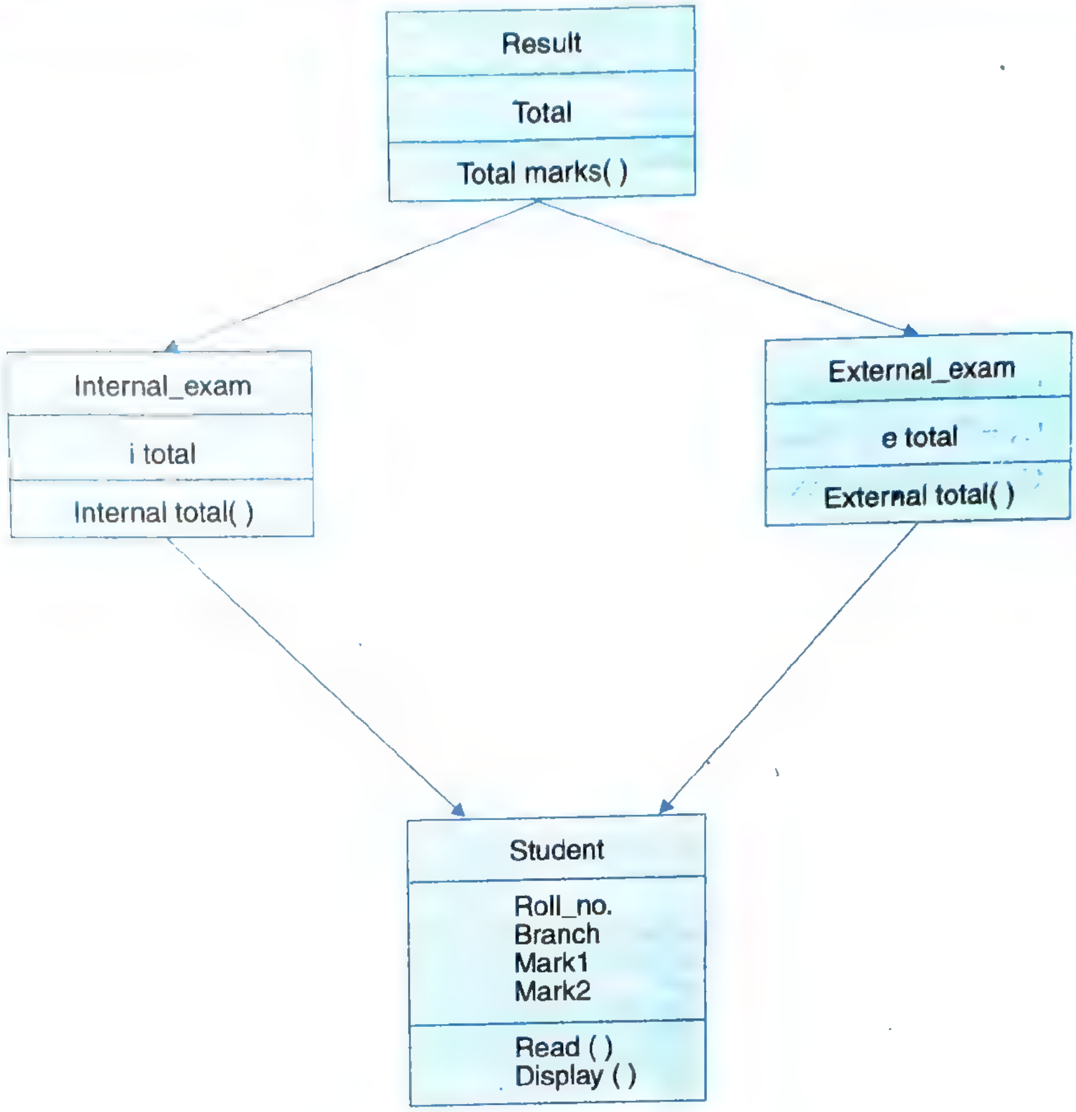


Fig. 6.18: Class diagram of result management system

(*b*) **Number of children (NOC):** The NOC is the number of immediate subclasses of a class in a hierarchy. In Fig. 6.18, NOC value for class student is 2.

(*c*) **Method inheritance factor (MIF):** It is system level metrics and is defined as :

$$\text{MIF} = \frac{\sum_{i=1}^{TC} M_i\,(C_i)}{\sum_{i=1}^{TC} M_a\,(C_i)}$$

where, $M_a(C_i) = M_i(C_i) + M_d(C_i)$

TC = total number of classes

$M_d(C_i)$ = Number of methods declared in a class

$M_i(C_i)$ = Number of methods inherited in a class.

A method is inherited if:

- It is defined in base class.
- It is visible in the subclass.
- It is not overridden in the subclass.

MIF is 0% for class lacking inheritance.

In Fig. 6.18, in student class, roll_no. and branch are private attributes whereas mark1 and mark2 are protected attributes.

$$TC = 4 \text{ (for figure 6.18)}$$

Let C_1 = student class

C_2 = Internal_Exam class

C_3 = External_Exam class

C_4 = Result class

$$MIF = \frac{M_i(C_1) + M_i(C_2) + M_i(C_3) + M_i(C_4)}{M_i(C_1) + M_i(C_2) + M_i(C_3) + M_i(C_4) + M_d(C_1) + M_d(C_2) + M_d(C_3) + M_d(C_4)}$$

$M_i(C_1)$ = Number of inherited methods in class student = 0

$M_i(C_2)$ = Number of inherited methods in class Internal_Exam = 2

Thus, $$MIF = \frac{0+2+2+2}{11} = \frac{6}{11}$$

(*d*) **Attribute inheritance factor (AIF).** It is defined as :

$$AIF = \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

where, $A_a(C_i) = A_i(C_i) + A_d(C_i)$

TC = total number of classes

$A_d(C_i)$ = number of attributes declared in a class

$A_i(C_i)$ = number of attributes inherited in a class

An attribute is inherited if:

- It is defined in the base class.
- It is visible in the subclass.
- It is not overridden in the subclass.

AIF is 0% for class lacking inheritance.

For the figure 12.7, TC = 4

Let C_1 = student

C_2 = Internal_Exam

C_3 = External_Exam

C_4 = Result

$$\text{AIF} = \frac{A_i(C_1) + A_i(C_2) + A_i(C_3) + A_i(C_4)}{A_i(C_1) + A_i(C_2) + A_i(C_3) + A_i(C_4) + A_d(C_1) + A_d(C_2) + A_d(C_3) + A_d(C_4)}$$

$A_i(C_1)$ = Number of inherited attributes in class student = 0

$A_i(C_2)$ = 2, as two attributes mark1 and mark2 are inherited by subclass Internal_Exam.

Similarly, $A_i(C_3) = 2$, $A_i(C_4) = 2$

$A_d(C_1) = 4$, as four attributes are declared in class student,

Similarly, $A_d(C_2) = A_d(C_3) = A_d(C_4) = 1$

Thus, $\text{AIF} = \frac{0+2+2+2}{13} = \frac{6}{13}$.

6.6 USE-CASE ORIENTED METRICS

The use-case diagrams and use cases have become very popular techniques during software requirements analysis and specifications. They also provide foundations for software design activities. Some of the metrics are:

6.6.1. Counting Actors

The simplest is the number of actors in a use case model. More the actors more complex would be the system under consideration. The actors may also be further categorized as simple, average or complex, depending on their activities. Shinji Kusumoto et. al. [KUSUO4] defined weighting factor for each type of actors and is given in Table 6.6.

Table 6.6: Actor weighting factors

Type	*Description*	*Factor*
Simple	Program interface	1
Average	Interactive or protocol driven interface	2
Complex	Graphical Interface	3

A simple actor represents another system with a defined Interface. An average actor is either another system that interacts through a protocol such as TCP/IP or it is a person interacting through a text based interface (such as an old ASCII terminal). A complex actor is a person interacting through a GUI interface. The actors weight can be calculated by adding these values together.

6.6.2 Counting Use Cases

Use case count is also very simple metric, but gives us idea about size and complexity of the software under consideration.

A use case is categorized as simple, average or complex. The basis of this decision is the number of transactions in a use case, including alternate paths. A simple use case has 3 or fewer transactions, an average use case has 4–7 transactions and a complex use case has more than 7 transactions as given in Table 6.7 [KUSUO4].

Table 6.7: Transaction-based weighting factors

Type	*Description*	*Factor*
Simple	3 or fewer transactions	5
Average	4 to 7 transactions	10
Complex	Mone than 7 transactions	15

The number of each use case type is counted in the software under consideration and then each number is multiplied by a weighting factor as shown in Table 6.7. Finally, use case weight is calculated by adding these values together.

This is a new area and more research is required to develop new matrics. The effort estimation using use cases is very important area however much work is required to be carried out to design a widely acceptable model.

6.7 WEB ENGINEERING PROJECT METRICS

World wide web (WWW) is expanding day-by-day and influencing every one of us very heavily and deeply. It will be interesting to know and design metrics for various aspects of web applications. Some of the web attributes are size of the web, its connectivity, visibility of sites and the distribution of information. Few metrics are given below :

6.7.1 Number of Static Web Pages

Many web applications use static pages. The user has no control over the contents of a static page. Normally, static pages are simple and easy to construst. This metric gives us the idea about size and complexity of the web application.

6.7.2 Number of Dynamic Web Pages

A dynamic page is different from a static page ; where user actions prepares the customised contents and display on the page. Hence contents are dependent on the action (s) of the user. Dynamic pages are very common in most of the web applications. These pages are more complex and require more effort than static pages. This metric also provides the idea about size, complexity and effort to construct the web application.

6.7.3 Number of Internal Page Links

The links are basically connections that provide a connectivity to some other web pages within the web application. If link count is more, complexity will be more and web page dependency factor will increase. It is expected to have less number of connections for a good web application.

6.7.4 Word Count

The word count metric is the total words on a page. This provides the idea about the contents on a page. This may range from low to extra high.

6.7.5 Web Page Similarity

Web page similarity metrics measure the extent of relatedness between two or more web pages. We may classify similarity metrics into content-based, link-based and usage based metrics. Content based similarity is measured by comparing the text of documents. Pages with similar contents may be considered related and designated in the same group. Link based measures rely on the hyperlink structure of a web graph to obtain related pages. Usage based similarity is based on patterns of document access. The intent is to group pages or even users into meaningful groups that can aid in better organisation and accessibility of websites.

6.7.6 Web Page Search and Retrieval

These are metrics for evaluating and comparing the performance of web search and retrieval services. These may be used to measure performance of search engines. One of the measure is time taken to search a web page and retrieval of desired information. The effectiveness of a search activity is also dependent on the quality of retrieved information. Although it is not easy to measure the quality of the content due to its dependency on the perceptions of a user.

6.7.7 Number of Static Content Objects

This includes static text based objects such as graphics, audio/video information, pictures in the web application. Many content objects may appear in a single web page.

6.7.8 Number of Dynamic Content Objects

The dynamic content objects are dependent on the actions of a user. This may include graphics, audio/video information, pictures etc. which are generated in a customised web application. Many content objects may appear in a single web page.

The design of web metrics is an upcoming area for characterizing and quantifying information on the web. We may see good number of metrics in the coming years to quantify various characteristics of web.

6.8 METRICS ANALYSIS

There is a wide range of techniques that we can use to assess internal attributes of the software product. All of these techniques produce data and, all this data can be expressed in numerical form. But collecting data should be seen as only the first stage of software assessment. We also have to analyze the data to make deductions about the quality of the software product and determine if it is sufficiently good to be released to customers.

The quantity of data we collect will often be quite large. Many textual, data structural and test coverage metrics are defined at the component level. If we collect, say 20 metrics for each component and we have 100 components in a system, then we will have 2000 data points, excluding metrics defined at the subsystem and system levels. It would be difficult to imagine

a software assessor simply gazing at pages of figures and rationally arriving at a pass/fail decision for that particular product. We need to use statistics to understand the numbers, to make deductions and then produce evidence to support those deductions, In many cases simply expressing the data in pie charts and histograms can reveal a great deal about the software. Nevertheless, if we wish to uncover the relationships between metrics to validate theories, methods like regression and correlation will be more appropriate. However, when applying any kind of statistics we need to be very careful. Software metrics data is often considered to be unusual in that it does not conform to the normally made assumptions on which many statistical tests and methods are based. However, this does not preclude the meaningful application of statistical techniques. We have many tests at our disposal, which can accommodate other distributional assumptions. It is therefore, important for the researcher to determine the specific statistical tests and methods needed for each analysis in turn. Furthermore, analyzing failure data needs a very particular type of statistics and there is a range of models, which are specifically used to predict future reliability. Nevertheless, there are limitations to the kind of predictions we can make about reliability [MART94].

6.8.1 Using Statistics for Assessment

The primary purpose of applying statistics to metrics data is to gain understanding. Therefore the choice of statistics we use will ultimately be determined by the audience for whom the statistics are intended. If we are providing an evaluation of a software product for project managers then generally we will want to use simple descriptive statistics. We can use simple graphs and tables to point out areas of high and low software quality and make comparisons with other projects or predefined target values.

We will also want to use the metrics data collected from different projects to define and refine the criteria on which the assessments are made. This means deciding the ranges of values that the metrics ought to have. Such an activity is essentially internal to the Quality Assurance department or test laboratory. In this case there is no need to restrict us to simple descriptive statistics, although these will still have a role. There are other more powerful techniques such as regression, correlation and multivariate techniques.

The advent of commercial statistical packages over the last few years has meant that people who have little or no knowledge of the underlying mathematics can readily use statistics. Many of the complex calculations that in the past had to be done manually or hard programmed into specific tools are no longer a concern. One particular facility provided by many such packages is the ability to generate graphs and diagrams automatically. Some packages are compatible with word processors so that these diagrams can be 'cut and pasted' into assessment reports.

The fact that statistical packages are so easy to use gives rise to the danger of applying inappropriate techniques. We still need to understand the assumptions on which a particular technique is based. If these are ignored then the conclusions are likely to be erroneous. It is all too easy to produce plausible-looking diagrams or correlations that are, in reality, totally meaningless.

Statistics is a large body of knowledge and it would certainly not be possible to describe all the methods that would conceivably be used with software metrics. We do describe here a range of techniques that have been specifically used in the field. These are:

[illegible] such as mean, median, maximum and minimum; graphical [illegible] such as histograms, pie charts and box plots. Principal components analysis [illegible] [illegible] regression and correlation techniques for [illegible] models for predicting future reliability — [illegible] from the other techniques.

[illegible] of these techniques can be used to interpret particular metrics [illegible] we can actually apply the statistics we explore why software [illegible] of data to which statistics have traditionally been applied.

[illegible] with Metrics Data

[illegible] make a number of assumptions about the data being analyzed. [illegible] in testing the significance of simple least-square regression [illegible] assumes that the data for both variables is at least [illegible] normally distributed. Both of these assump- [illegible] we consider and so we should check carefully [illegible]

Normal Distribution

[illegible] the assumption that the data under analysis is drawn from [illegible] The frequency distribution for normal data has a bell-shaped [illegible]

[illegible] measurement data have been shown to follow this kind of distri- [illegible] many techniques. However, this is often not the case for [illegible]

Fig. 8.18: The normal distribution

This has implications when considering testing for equivalence of mean averages from different samples. When using the t-test for this purpose it is necessary that the sample variable is normally distributed. We can also circumvent this problem by using robust techniques. These robust techniques (often called non-parametric) require few distributional assumptions or perform just as well even when their assumptions are violated.

Outliers

An outlier is a data point which is outside the normal range of values of a given population. In non-software data, outliers are often due to errors in measurement or are caused by systematic

bias. For this reason they are frequently removed from the data set and subsequently ignored. In software metrics data, however, the outliers are often the most interesting data points. For example, if we are measuring component size, it is not uncommon to have a single software component, which is 10–20 times larger than any other component. It is precisely these large components, which may give rise to problems in maintainability and so they should certainly not be removed from the data set.

Measurement scale

The scale on which metrics are defined will generally determine which statistics can be meaningfully applied. Most physical measures are ratio, *e.g.*, length, mass, voltage, pressure. Therefore people would be forgiven for assuming the same about software metrics. This is often not the case.

To decide which scale a particular metric is defined on, we need to go back to the actual definition of the metric and reason about the relationship imposed by the attribute being measured.

Multicollinearity

Many multivariate techniques such as multivariate regression require that the variables (*i.e.*, metrics) are independent of each other. Independent in this sense means that they do not correlate with one another. Unfortunately most static flow graph and textual metrics are correlated with size. It does not mean that these metrics all measure size-they measure many different attributes like number of decisions or nesting-but these attributes tend to be highly correlated with size and by implication with each other. This is because a component with a high number of decisions or a high level of nesting is also likely to be large. The same phenomenon of multicollinearity exists for coverage metrics, so branch coverage will be highly correlated with statement coverage, DDP coverage, etc.

The solution to this is to use either factor analysis (FA) or principal components analysis (PCA) to reduce the set of metrics to a smaller set of independent components. FA usually requires the data to be normally distributed, which, as we know, is rarely going to be true. PCA on the other hand is a robust technique because it relies on no distributional assumptions.

PCA is applied to data in order to reduce the number of measures used and to simplify correlation patterns. It attempts to solve these problems by reducing the dimensionality represented by these many related variables to a smaller set of principal components while retaining most of the variation from the original variables. These new principal components are uncorrelated and can act as substitutes for the original variables with little loss of information.

6.8.3 The Common Pool of Data

We often want to compare metrics from one software product with typical values observed in many other products. This requires creating a common pool of data from previous projects. If metrics are defined at the system level then we will have one value per software project. However, metrics defined at finer levels of granularity will contribute to the common pool in varying degrees. The pool of metric values defined at the component level will be influenced

more by those software projects, which have a greater number of components. When establishing the common pool we need to be aware of three points.

1. The selection of projects should be representative and not all come from a single application domain or development styles. The exception is when a QA department with a fixed range of developments or a defined process uses the common pool.
2. No single very large project should be allowed to dominate the pool; this is less likely as the number of projects increases.
3. For some projects, certain metrics may not have been collected; we should ensure this is not a source of bias.

Only when these three points are satisfied can we be sure about making comparisons with the common pool.

6.8.4 A Pattern for Successful Applications

Successful applications of metrics abound but are not much talked about in the public literature. Mature metrics can help us to predict expected number of latent bugs, help us to decide how much testing is enough and how much design effort, cost, elapsed time, and all the rest we expect from metrics. Here's what it takes to have a success [BEIZ90].

1. *Any Metric Is Better Than None:* Use simplest possible metric like weight of program listings first and then implement "token counting" and "function counting" as the next step. Worry about the fancy metrics later.

2. *Automation Is Essential:* Any metrics project that relies on having the developers fill out long questionnaires or manually calculate metric values is doomed. They never work. If they work once, they won't the second time. If we've learned anything, it's that a metric whose calculation isn't fully automated isn't worth doing; it probably harms productivity and quality more than any benefit we can expect from metrics.

3. *Empiricism Is Better Than Theory:* Theory is at best a guide to what makes sense to include in a metrics project — there's no theory sufficiently sound today to warrant the use of a single, specific metric above others. Theory tells us what to put into the empirical pot. It's the empirical, statistical data that we must use for guidance.

4. *Use Multifactor Rather Than Single Metrics:* All successful metrics programs use a combination (typically linear) of several different metrics with weights calculated by regression analysis.

5. *Don't Confuse Productivity Metrics with Complexity Metrics:* Productivity is a characteristic of developers and testers. Complexity is a characteristic of programs. It's not always easy to tell them apart. Examples of productivity metrics incorrectly used in lieu of complexity metrics are: number of shots it takes to get a clean compilation, percentage of components that passed testing on the first attempt, number of test cases required to find the first bug. There's nothing wrong with using productivity metrics as an aid to project management, but that's a whole different story. Automated or not, a successful metrics program needs developer cooperation. If complexity and productivity are mixed up, be prepared for either or both to be sabotaged to uselessness.

6. *Let Them Mature:* It takes a lot of projects and a long time for metrics to mature to the point where they're trustworthy. If the typical project takes 12 months, then it takes ten to fifteen projects over a 2–3 year period before the metrics are any use at all.

7. *Maintain Them:* As design methods change, as testing gets better, and as QA functions to remove the old kind of bugs, the metrics based on that past history lose their utility as a predictor of anything. The weights given to metrics in predictor equations have to be revised and continually reevaluated to ensure that they continue to predict what they are intended to predict.

8. *Let Them Die:* Metrics wear out just like test suites do, for the same reason—the pesticide paradox. Actually, it's not that the metric itself wears out but that the importance we assign to the metric changes with time. We have seen projects go down the tubes because of worn-out metrics and the predictions based on them.

REFERENCES

[ALBR79] Albrecht A., "*Measuring Application Development Productivity*", Proc. IBM Application Development Symposium, Monterey, California Oct 14–17, 1979.

[ALBR83] Albrecht A., and Gaffney J.E., "*Software Function Source Lines of Code and Development Effort Prediction: A Software Science Validation*", IEEE Trans. Software Engineering, SE-9 639–648, 1983.

[ARVI06] Arvinder Kaur, "*Development of Techniques for Good Quality Object Oriented Software*", Ph.D. Thesis, University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India 2006.

[AGGA94] Aggarwal K.K. and Yogesh Singh, "*A Modified Approach for Software Science Measures*", ACM SIGSOFT Software Engineering Notes, USA, July, 1994.

[BACH90] Bache and Monica, "*Measures of Testability as a Basis for Quality Assurance*", Software Engineering Journal, March, PP-86–92, 1990.

[BAIL81] Bailey J.W., and Basili V.R.," *A meta Model for Software Development Resource Expenditures*", Proc. of the Int. Conf. on Software Engineering, 107–116, 1981.

[BASI75] Basili V.R. and Turner A.J., "*Iterative Enhancement: a Practical Technique for Software Development,*" IEEE Trans. On Software Engg., SE-1, 390–396, Dec. 1975.

[BEIZ90] Boris Beizer, "*Software Testing Techniques*", Van Nostrand Reinhald International Co. Ltd., UK, 1990.

[BOEH81] Boehm B., "*Software Engineering Economics*", Prentice-Hall, Englewood Cliffs, NJ. 1981.

[BRUC92] Bruci I. Blum,"*Software Engineering — A Holistic View*", Oxford University Press, NY, 1992.

[BULU73] Bulut N., "*Invariant Proprties of Algorithms*", Ph.D. Thesis, Purdue University, August, 1973.

[CARD87] Card D.N. and Agresti W.W., "*Resolving Software Science Anomoly*", Journal of Systems and Software, Vol.7, 29–35, 1987.

[CHHA01] Chhabra Jitender Kumar, Aggarwal K.K., Yogesh Singh, "*Computing Program Weakness using Module Coupling*", ACM SIGSOFT Software Engineering Notes, Vol. 27, No. 1, January, 63–66, 2002.

[CONT86] Conte S.K., Dunsmore H.E., Shen V.Y., "*Software Engineering Metrics and Models*", The Benjamin/Cummings Pub. C. Inc., California, USA, 1986.

[CHHA2K] Chhabra Jitender Kumar, Dinesh Chutani, Aggarwal K.K., Yogesh Singh, "*Effect of Data Coupling on Program Weakness*", International Conference on Quality, Reliability and IT at the Turn of the Millennium, New Delhi, Dec. 2000.

[DUNS79] Dunsmore H.E. and Gannon J.D., "*Analysis of The Effects of Programming Factors on Programming Effort*", Journal of systems and software, 141–153, 1980.

[EJIO91] "*Software Engineering with Formal Metrics*", QED Information Sciences, Wellesley, Massachusetts 1991.

[ELSH76] Elshoff J.L., "*An Analysis of Some Commercial PI/1 Programs*", IEEE Transactions on Software Engineering", SE-2, 113–120, June, 1976.

[FENT04] N.E. Fenton, "*Software Metrics* (2 W)" Thomson Books, 2004.

[FITZ78] Fitzisimmory A. and Love T., "*A Review and Evaluation of Software Science*", ACM Computing Surveys, Vol 10, March, 1978.

[GHEZ94] Carlo Ghezzi et. al. "*Software Engineering*", PHI, 1994.

[GOOD93] Paul Goodman, "*Practical Implementation of Software Metrics*", McGraw Hill Book Company, UK, 1993.

[HALS77] Halstead M.H., "*Elements of Software Science*", New York, Elsevier North Holland, 1977.

[HAME85] Hamer P. and Frewin G., "*Software Metrics: A Critical Overview*", Pergamon Infotech State of the art report 13 (2), 1985.

[HENR81] Henry S. and Kafura D., "*Software Structure Metrics Based on Information Flow*", IEEE Trans. on Software Engineering SE-7, 5, 510–518, Sept 1981.

[INCE89] Ince D., "*Software Metrics: Measurement for Software Control and Assurance*", New York: Elsevier, 1989.

[JAME78] Elshoff J.L., "*An Inrestigation in to the Effects of the Counting Method Used on Software Science Measurements*," ACM SIGPLAN Notices, Vol 13, 30–45, Feb, 1978.

[JENS85] Jensen H.A. and Vairavan K., "*An Experimental Study of Software Metrics for Real Time Software*", IEEE Trans, on Software Engineering, 231–234, Feb, 1985.

[KITC90] Kitchenham B., "*Empirical Studies of Assumption Underlying Software Cost Estimation Models*", Proc. of European COCOMO User Group, 1990.

[KUSU04] S. Kusumoto et. al., "*Estimating Effort by Use Case Points: Method, Tool and Case Study*", Proc. of the 10th International Symposium on Software Metrics, 1530–1435/04, 2004.

[LI93] W Li and S. Henry "*Object Oriented Metrics that Predict Maintainability*", Journal of Systems Software, 23, 111—122, 1993.

[MART94] Martin Neh, "*Software Metrics for Product Assessment*", McGraw Hill Book Co., UK, 1994.

[MATS94] Matson E.M., et al., "*Software Development Cost Estimation Using Function Points*", IEEE Trans. on software Engineering", Vol 20, No.4. April, 1994.

[MEHN86] Mehnadiralta B., and Grover P.S., "*Measuring Computer Programs*", Proc. of CSI Annual Convention, India, 1986.

[RAMA88] Ramamurthy B. and Melton A., "*A Synthesis of Software Science Measures and the Cyclomatic Number*," IEEE Trans. on Software Engineering Vol. 14. No.8, August, 1988.

[ROSE97] Rosenberg L.H., "*Applying and Interpreting Object Oriented Metrics*", SATC Project of NASA on Object Oriented Matrics, USA, 1997.

[RUBE68] Rubey R.J. and R.D. Hartwick, "*Quantitative Measurement of Program Quality*", Proc. ACM Nat. Conf. PP 671–677, 1968.

[SESH61] Sheshu S. and Recd M.B., "*Linear Graphs and Electrical Networks*", Addision Wesely, USA, 1961.

[SHEN83] Shen V.Y., Conte S.D., and Dun Smore H.E." *Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support*", IEEE Trans. on Software Engg., Vol. SE-9 No.2., 1983,155–165, March, 1983.

[STEP95] Stephen Treble and Neil Douglas, "*Sizing and Estimating Software in Practice*", McGraw Hill Book Company, London, 1995.

[SHYA91] Shyam R. Chidamber and Kemerer C.F., "*Towards a Metrics Suite for Object Oriented Design*", ACM OOPSLA, 171–211, 1991.

[SHYA94] Shyam R. Chidamber and Kemerer C.F., "*A Metrics Suite for Object Oriented Design*", IEEE Transactions on Software Engineering, vol. 20, No. 6, 476-493, June 1994.

[STRO67] Stroud J.M., "*The Fine Structure of Psychological Time*", Annuals of New York Academy of Science 138, 2, 623–631, 1967.

[THEB83] Thebaut S.M., "*The Saturation Effect in Large Scale Software Development Its Impact and Control*", Ph.D. Thesis, Department of Computer Science, Purdue University, West Lafeyette, IN, May, 1983.

[TRAC88] Tracz W., "*Software Reuse Emerging Technologies*", IEEE Computer Society Press, Washington DC, 1988.

[YOGE95] Singh Yogesh, "*Metrics and Design Techniques for Reliable Software*", Ph.D Thesis, Kurukshetra University, Kurukshetra (India) , July, 1995.

[YOGE98] Singh Yogesh and Pradeep Bhatia, "*Module Weakness — A New Measure*", ACM SIGSOFT Software Engineering Notes, 81, July 1998.

MULTIPLE CHOICE QUESTIONS

Note: *Choose most appropriate answer of the following questions.*

6.1. Which one is not a category of software metrics ?

(*a*) product metrics (*b*) process metrics

(*c*) project metrics (*d*) people metrics.

6.2. Software science measures are developed by

(*a*) M. Halstead (*b*) B. Littlewood

(*c*) T.J. McCabe (*d*) G. Rothermal.

6.3. Vocabulary of a program is defined as:

(*a*) $\eta = \eta_1 + \eta_2$ (*b*) $\eta = \eta_1 - \eta_2$

(*c*) $\eta = \eta_1 \times \eta_2$ (*d*) $\eta = \eta_1 / \eta_2$.

6.4. In Halstead theory of software science, volume is measured in bits. The bits are

(*a*) number of bits required to store the program

(*b*) actual size of a program if a uniform binary encoding scheme for vocabulary is used

(*c*) number of bits required to execute the program

(*d*) none of the above.

6.5. In Halstead theory, effort is measured in

(*a*) parson-months (*b*) hours

(*c*) elementary mental discriminations (*d*) none of the above

6.6. Language level is defined as

(a) $\lambda = L^3V$ (b) $\lambda = LV$

(c) $\lambda = LV^*$ (d) $\lambda = L^2V$

6.7. Program weakness is

(a) WM = $\overline{LV} \times \gamma$ (b) WM = $\overline{LV} / \gamma$

(c) WM = $\overline{LV} + \gamma$ (d) none of the above.

6.8. 'FAN IN' of a component A is defined as

(a) count of the number of components that can call, or pass control, to component A

(b) number of components related to component A

(c) number of components dependent on component A

(d) none of the above.

6.9. 'FAN OUT' of a component A is defined as

(a) number of components related to component A

(b) number of components dependent on component A

(c) number of components that are called by component A

(d) none of the above.

6.10. Which is not a size metric?

(a) LOC (b) function count

(c) program length (d) cyclomatic complexity.

6.11. Which one is not a measure of software science theory?

(a) vocabulary (b) volume

(c) level (d) logic

6.12. A human mind is capable of making how many number of elementary mental descriminations per second (i.e., stroud number)?

(a) 5–20 (b) 20–40

(c) 1–10 (d) 40–80

6.13. Minimal implementation of any algorithm was given the following name by Halstead:

(a) volume (b) potential volume

(c) effective volume (d) none of the above.

6.14. Program volume of a software product is

(a) $V = N \log_2 n$ (b) $V = (N/2) \log_2 n$

(c) $V = 2N \log_2 n$ (d) $V = N \log_2 n + 1$

6.15. Which one is the international standard for size measure?

(a) LOC (b) function count

(c) program length (d) none of the above.

6.16. Which one in not on object oriented metric ?

(a) RFC (b) CBO

(c) DAC (d) OBC.

6.17. Which metric also considers indirect connected methods ?

(a) TCC (b) LCC

(c) both of the above (d) none of the above.

6.18. Depth of inheritance tree (DIT) can be measured by:

(*a*) number of ancestor classes (*b*) number of successor classes

(*c*) number of failure classes (*d*) number of root classes.

6.19. A dynamic page is:

(*a*) where contants are not dependent on the actions of the user.

(*b*) where contents are dependent on the actions of the user

(*c*) where contents cannot be displayed

(*d*) none of the above.

6.20. Which one is not a size measure ?

(*a*) LOC

(*b*) FP

(*c*) cyclomatic complexity

(*d*) program length.

EXERCISE

6.1. Define software metrics. Why do we really need metrics in software?

6.2. Discuss the areas of applications of software metrics. What are the problems during implementation of metrics in any organization ?

6.3. What are various categories of software metrics ? Discuss with the help of suitable examples.

6.4. Explain the Halstead theory of software science. Is it significant in today's scenario of component based software development ?

6.5. What is the importance of language level in Halstead theory of software science?

6.6. Give Halstead's software science measures for:

(*i*) program length (*ii*) program volume

(*iii*) program level (*iv*) effort

(*v*) language level.

6.7. For a program with number of unique operators $\eta_1 = 20$ and number of unique operands $\eta_2 = 40$, compute the following:

(*i*) program volume (*ii*) effort and time

(*iii*) program length (*iv*) program level.

6.8. Develop a small software tool that will perform a Halstead analysis on a programming language source code of your choice.

6.9. Write a program in C and also PASCAL for the calculation of the roots of a quadratic equation. Find out all software science metrics for both the programs. Compare the outcomes and comment on the efficiency and size of both the source codes.

6.10. How should a procedure identifier be considered, both when declared and when called? What about the identifier of a procedure that is passed as a parameter to another procedure?

6.11. It is interesting to examine how the ratio $(N-\hat{N})/N$ varies when a program is divided into parts. Actually, some experiments show that the partitioning of vocabularies due to program modularization maintains the stability of the ratio $(N-\hat{N})/N$. Two extreme situations may occur:

a. η_1 and η_2 are the same for all parts.

b. η_1 and η_2 are partitioned into disjoint subsets by modularization.

Compute the variations of $\hat{N}$ for a program with $N = \hat{N} = 72$, $\eta_1 = 4$ and $\eta_2 = 16$ when the program is divided into two parts, under the two extreme assumptions.

Warning: It may be difficult to divide a program in such a way so as to satisfy the latter assumption. (At least one procedure definition and call must share an identifier.) For large values of the quantities involved, however, we can assume that, at least, η_1 and η_2 have small intersections with respect to their size [GHEZ 94].

6.12. Define data structure metrics. How can we calculate amount of data in a program ?

6.13. Describe the concept of module weakness. Is it applicable to programs also ?

6.14. Write a program for the calculation of roots of a quadratic equation. Generate cross reference list for the program and also calculate $\overline{LV}$, γ and WM for this program.

6.15. Show that the value of SP at a particular statement is also the value of LV at that point.

6.16. Discuss the significance of data structure metrics during testing.

6.17. What are information flow metrics ? Explain the basic information flow model.

6.18. Discuss the problems with metrics data. Explain two methods for the analysis of such data.

6.19. Show why and how software metrics can improve the software process. Enumerate the effect of metrics on software productivity.

6.20. Why does lines of code (LOC) not measure software nesting and control structures?

6.21. Several researchers in software metrics concentrate on data structure to measure complexity. Is data structure a complexity or quality issue, or both?

6.22. List the benefits and disadvantages of using Library routines rather than writing own code.

6.23. Compare software science measures and function points as measures of complexity. Which do you think more useful as a predictor of how much particular software's development will cost?

6.24. Some experimental evidence suggests that the initial size estimate for a project affects the nature and results of the project. Consider two different managers charged with developing the same application. One estimates that the size of the application will be 50,000 lines, while the other estimates that it will be 100,000 lines. Discuss how these estimates affect the project throughout its life cycle.

6.25. Which one is the most appropriate size estimation techinque and why?

6.26. Discuss the object oriented metrics. What is the importance of metrics in object oriented software development?

6.27. Define the following:
RFC, CBO, DAC, TCC, LCC and DIT.

6.28. What is the significance of use case metrics? Is it really important to design such metrics?